EXPLORATION AND COMBAT IN NETHACK

by
Jonathan C. Campbell

School of Computer Science
McGill University, Montréal

Monday, April 16, 2018

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

Abstract

Roguelike games share a common set of core game mechanics, each complex and involving randomization which can impede automation. In particular, exploration of levels of randomized layout is a critical and yet often repetitive mechanic, while monster combat typically involves careful, detailed strategy, also a challenge for automation. This thesis presents an approach for both exploration and combat systems in the prototypical roguelike game, *NetHack*. For exploration we advance a design involving the use of *occupancy maps* from robotics and related works, aiming to minimize exploration time of a level by balancing area discovery with resource cost, as well as accounting for the existence of secret access points. Our combat strategy involves the use of *deep Q-learning* to selectively match player items and abilities to each opponent. Through extensive experimentation with both approaches on NetHack, we show that each outperforms simpler, baseline approaches. Results are also presented for a combined combat-exploration algorithm on a full NetHack level context. These results point towards better automation of such complex roguelike game environments, facilitating automated testing and design exploration.

Résumé

Les jeux du genre *roguelike* partagent un ensemble commun de systèmes de jeu de base. Ces systèmes sont souvent complexes et impliquent une randomisation qui peut être un obstacle à l'automatisation. En particulier, l'exploration de niveaux générés aléatoirement est un mécanisme critique et pourtant souvent répétitif, tandis que le combat contre les monstres nécessite généralement une stratégie méticuleuse, ce qui constitue également un défi pour l'automatisation. Cette thèse présente une approche pour les systèmes d'exploration et de combat dans le jeu prototypique roguelike, *NetHack*. Pour l'exploration, nous présentons un algorithme qui utilise des *occupancy maps* provenant du domaine de la robotique, visant à minimiser le temps d'exploration d'un niveau en équilibrant la découverte de cartes avec le coût des ressources et en tenant compte des points d'accès secrets. Notre stratégie de combat implique l'utilisation du *deep Q-learning* pour faire correspondre de manière sélective les objets et les capacités d'un joueur à chaque adversaire. Grâce à une expérimentation approfondie des deux approches sur NetHack, nous montrons que chacun surpasse des approches de base plus simples. Nous présentons également des résultats pour un algorithme de combat/exploration combiné qui peut jouer une version relativement complète de NetHack. Ces résultats indiquent une meilleure automatisation de ces environnements de jeu complexes, facilitant les tests automatisés et l'exploration de l'espace de conception.

Acknowledgements

I would like to express my deepest gratitude and appreciation for the immeasurable support given to me by my supervisor, Professor Clark Verbrugge. Prof. Verbrugge has given me invaluable advice at every step of the way on this academic journey and this work is most assuredly significantly better thanks to his help. I would also like to thank my parents, who have always supported me.

Table of Contents

List of Figures

Chapter 1
Introduction

Video games often involve a vast array of complexity, arising from the exposition of game systems which provide challenge and fun to players. Systems can be low-level basic mechanics, like movement or dice rolling, or more complicated constructs, such as level map exploration or monster combat, which require more thinking and planning for a player to master. At the same time, this complexity also poses problems for automated gameplay.

One particular genre of games known as *roguelikes* typically feature in common several core systems, including movement, exploration, combat, items, and resource management. Key to this genre is the idea of randomization: each time a roguelike game is played, the gameplay experience is different from past playthroughs because of randomly-generated level layouts, item properties, monsters, or other features. This randomization further adds to the complexity involved with making automated approaches.

With its common game systems and concept of randomization, the roguelike game genre provides an interesting and novel environment to experiment with approaches towards automating said game systems. This thesis will examine and attempt to provide algorithmic approaches to two key roguelike systems: exploration of a game level and combat with monsters. The prototypical roguelike game *NetHack*, with extensive versions of these two systems, will be used as experiment environment. Below we introduce and motivate the analyses of the exploration and combat systems.

1.1 Exploration

Exploration in roguelike games is essential to game progress and resource acquisition. The high level of repetition involved, however, makes it useful to automate the exploration process, both as an assistance in game design as well as for relieving player tedium in relatively safe levels or under casual play. It is also useful to streamline control requirements for those operating with reduced interfaces [Sut17]. Basic forms of automated exploration are found in several roguelikes, including the popular *Dungeon Crawl Stone Soup*.

Algorithmic approaches to exploration typically aim at being exhaustive. Even with full prior information, however, ensuring complete coverage of a level can result in significant inefficiency, with coverage improvement coming at greater costs as exploration continues [CV16]. Diminishing returns are further magnified in the presence of secret (hidden) rooms, areas which must be intentionally searched for at additional, non-trivial resource cost, and which are a common feature of roguelike games. In such contexts the complexity is driven less by the need to be thorough and more by the need to balance the time spent exploring a space with respect to the amount of benefit accrued (area revealed, items collected).

In this thesis a novel algorithm for exploration of an initially unknown environment is presented. The design aims to accommodate features common to roguelike games, focusing in particular on an efficient, balanced approach to exploration which considers the cost of further exploration in relation to the potential benefit. The relative importance of different areas is factored in, prioritizing coverage and discovery of rooms versus full/corridor coverage, as well as dealing with discovery of secret areas. The design is intended to provide a core system useful in higher level approaches to computing game solutions, as well as in helping good game design. For the former we hope to reduce the burden of exploration itself as a concern in research into techniques that fully automate gameplay.

The approach takes root from the *occupancy map* data structure used in robotics, combined with concepts from a variation of said structure used in a video game target searching algorithm [Isl13]. Planning in the algorithm considers both distance and utility of unexplored areas. In this way we can control how the space is explored, following a probability gradient that flows from places of higher potential benefit.

We compare this approach with a simpler, greedy algorithm more typical of a basic automated strategy, applying both to levels from NetHack. The NetHack environment gives us a realistic and frequently mimicked game context, with uneven exploration potential (rooms versus corridors), critical resource limitations (every move consumes scarce food resources), and a non-trivial, dungeon-like map environment, including randomized placement and discovery of secret doors. Compared to the greedy approach, our algorithm shows improvement in overall efficiency, particularly with regard to discovery of secret areas. We enhance this investigation with a deep consideration of the many different parameterizations possible, showing the relative impact of a wide variety of algorithm design choices.

Specific contributions of this section include:

- We heavily adapt a known variation on occupancy maps to the task of performing efficient exploration of dungeon-like environments. Our design represents a significant deviation from the basic searching process for which the technique was originally designed.
- We further extend the exploration algorithm to address the presence of secret doors. Locating and stochastically revealing an unknown set of hidden areas adds notable complexity and cost to optimizing an exploration algorithm.
- Our design is backed by extensive experimental work, validating the approach and comparing it with a simpler, greedy design, as well as exploring the impact of the variety of different parameterizations available in our approach.

1.2 Combat

In many game genres, the mechanic of combat can require non-trivial planning, selecting attack and defense strategies appropriate to the situation, while also managing resources to ensure future combat capability. This arrangement is particularly and notoriously true of the roguelike genre, which often features a wide range of weaponry, items, and abilities that have to be well-matched to an also widely varied range of opponents. Such diversity becomes an interesting and complex problem for game AI, requiring a system to choose

among an extremely large set of actions, which likewise can be heavily dependent on context. As combat behaviours rest on basic movement control and state recognition, the combined problem poses a particularly difficult challenge for learning approaches where learning costs are a factor.

This thesis will describe a machine learning approach addressing one-on-one, player versus monster combat in NetHack. We focus on the core combat problem, applying a deep learning technique to the basic problem of best selecting weapons, armor, and items for optimizing combat success. To reduce complexity and accelerate the learning process, we build on a novel, abstracted representation of the game state and action set. This representation limits generality of the AI, but allows it to focus on learning relevant combat strategy, applying the right behaviour in the right context, while relying on well-understood algorithmic solutions to lower-level behaviours such as pathing.

We evaluate our learning-based model on three scenarios, each increasing in difficulty, and demonstrate its improvement over a simple (scripted) baseline combat strategy through its ability to learn to choose appropriate weaponry and take advantage of inventory contents.

A learned model for combat eliminates the tedium and difficulty of hard-coding responses for each monster and player inventory arrangement. For roguelikes in general, this breadth is a major source of complexity, and the ability to learn good responses opens up potential for AI bots to act as useful design agents, enabling better game tuning and balance control. Further automation of player actions also has advantage in allowing players to confidently delegate highly repetitive or routine tasks to game automation.

Specific contributions of this section include:

- We describe a basic deep learning approach to a subset of NetHack combat. Our design abstracts higher-level actions to accelerate learning in the face of an otherwise very large low-level state space and action set.

- We apply our design to three NetHack combat contexts, considering battle against one single (fixed) monster in a limited form and battle against a variety of single monsters with a full inventory of combat-related items.

- Experimental work shows the learned result is effective, generally improving over a

simple, scripted baseline. Detailed examination of learned behaviour indicates appropriate strategies are selected.

The combat approach will be further discussed in Chapter 4, while the exploration algorithm will be discussed in Chapter 3. We also attempt to combine these two components into a general automated strategy for NetHack, presented in Chapter 5. Further background on the approaches is found in Chapter 2, while related work is outlined in Chapter 6.

1.3 Papers published

Portions of this work were previously published in two conference publications. One poster paper of four pages described the general exploration algorithm [CV17a], while one full paper of six pages described the combat approach [CV17b]. I was the primary contributor to both papers, and co-authored both with my supervisor, Prof. Clark Verbrugge.

Chapter 2
Background

In this chapter the background to the work proper will be discussed: namely, the game in which the experiments take place (NetHack) including cursory foci on its relevant subsystems, as well as sections on occupancy maps and reinforcement learning, the two main technologies used.

2.1 NetHack

In this section we will discuss *NetHack*, a video game of the roguelike genre created in 1987 and maintained to this day by a core team of developers. It was and still is a very popular game, and is now considered to be a prototypical game of the genre. Gameplay occurs on a two-dimensional text-based grid of size 80x20, wherein a player can move around, collect items, fight monsters, and travel to deeper dungeon levels. We will begin with a brief overview of the game, followed by a study of the subsystems relevant to this thesis (exploration/combat systems), and end by discussing how we interface with the game. An example of a NetHack level is shown in Figure 2.1.

The game is a difficult one. To win, a player must travel through all 53 levels of the dungeon, emerge victorious against the high priest of Moloch and collect the Amulet of Yendor, then travel back up through all the levels while being pursued by an angry Wizard and finally ascend through the five elemental planes [NetHack Wiki16b]. Although detailed statistics for the game are unavailable, on at least one server where NetHack can

be played, total ascension (win) rate was only 0.605% of over one million games as of mid-2017 [pNs16].

NetHack has all the features of the canonical roguelike game: random generation of levels and other game features, permanent death, a turn-based system, grid-based map, important resource management considerations, combat with monsters, and exploration and discovery of the game world (this list according to the Berlin Interpretation of roguelikes [Int08]). With all of these different systems and the interplay between them, as well as the high level of difficulty involved, NetHack is quite a complex game, yet has not been widely studied in an academic context. We detail below the concepts relevant to the present work: namely, NetHack levels and their exploration, resource concerns, and monster combat.

2.1.1 Exploration in NetHack

Exploration is perhaps the most important mechanic in NetHack – the majority of a player's actions will be devoted to discovering the layout of the current dungeon level by moving around. Here we discuss the basic idea of exploring NetHack levels and why it is important to explore (i.e., discover rooms) in an efficient manner. We then discuss the presence of secret areas in NetHack, initially hidden rooms that require further action to discover.

Levels in NetHack consist of these rooms, which are large, rectangular and connected by maze-like corridors. There are typically around eight rooms in a dungeon level. Levels can be sparse, with many empty (non-traversable) tiles. For the most part, levels are created using a procedural content generator (thus allowing our algorithms to be tested on a wide variety of different map configurations). An example of a typical NetHack map is presented in Figure 2.1, and other maps can be seen in Figures 2.2, 3.1, and 3.3. At the start of each level, the player can observe only the contents, walls and doors of the room they have spawned in, and must explore to uncover more of the map (e.g., by going through a door and corridors to another room).

Although map exploration is important, it is also exigent to do so in a minimal fashion. Movement in NetHack is turn-based (each move taking one turn), and the more turns made, the more hungry one becomes. Hunger can be satiated by food, which is randomly and

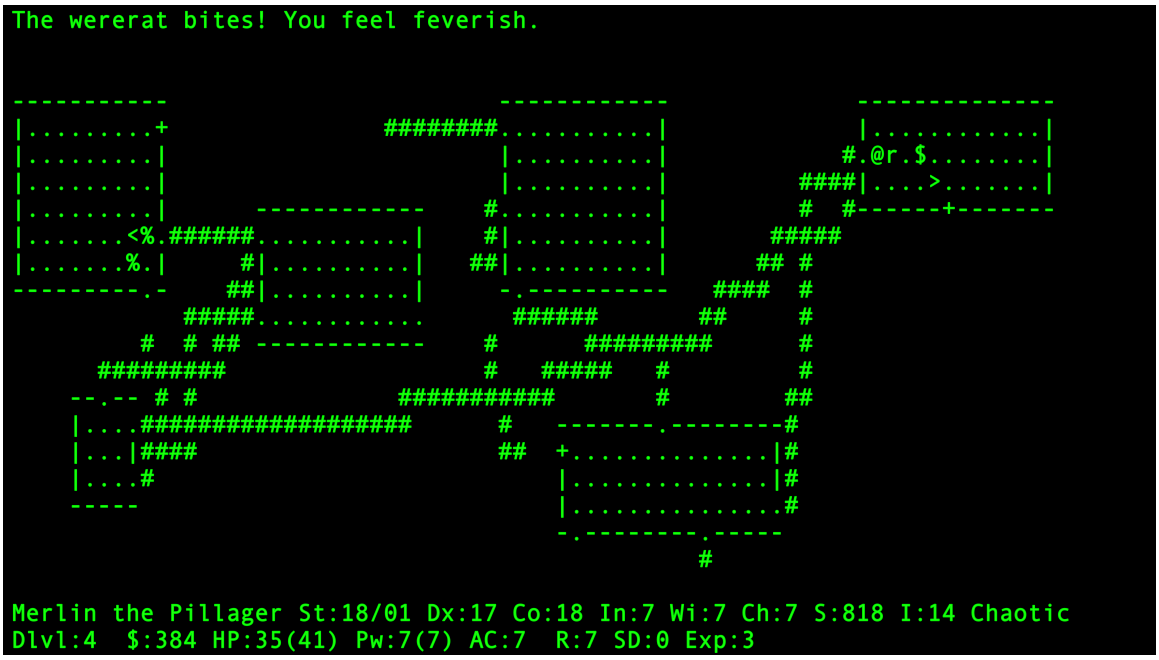

Figure 2.1 A game of NetHack with the player ('@' character, currently in the top-right room) in battle against a were-rat ('r' character) on level 4 of the dungeon. A typical NetHack map is composed of corridors ('#') that connect rectangular rooms. Room spaces ('.') are surrounded by walls ('|' and '-'), and unopened doors ('+'), which might lead to other, currently unseen rooms, or cycle around to already visited ones. Monsters like the were-rat are mostly represented by alphabetical characters, while items are represented by other miscellaneous characters. The bottom two lines contain information about the player's current attributes and statistics, while the top line displays a brief summary or narrative of what has occurred in the previous turn.

sparingly placed within the rooms of a level [NetHack Wiki16a]. Most food does not regenerate after having been picked up, so a player must move to new levels at a brisk pace to maintain food supplies. A player that does not eat for an extended period will eventually starve to death and lose the game [NetHack Wiki15]. Many a player has had their game cut cruelly short in this way.

With this in mind, it is critical to minimize the number of actions taken to explore a level so that food resources are preserved. Rooms are critical to visit since they may contain food and items that increase player survivability, and one random room will always contain the exit to the next level, which must be discovered to advance further in the game. Conversely, corridors that connect rooms have no intrinsic value. Some may lead to dead-ends or circle

around to already visited rooms. Exploring all corridors of a level is typically considered a waste of valuable actions. Therefore, a good exploration strategy will minimize visitation of corridors while maximizing room visitation.

2.1.1.1 Secret rooms in NetHack

Secret areas are a popular element of game levels and motivate comprehensive exploration of a space. These areas are not immediately observable by a player but must be discovered through extra action on the player's part. Secret areas are present in NetHack; their generation and discovery action are detailed below.

Secret areas are common in roguelike games in particular, and tend to be procedurally generated along with the rest of the general map layout. This procedural generation tends to lessen the reward associated with the area: in NetHack, the contents of secret rooms are the same as in regular, non-secret rooms, potentially including items, food, or level exit. However, discovery of secret areas is still important for these occasional items, and in some instances crucial, if the exit happens to be generated in one, or if a large chunk of the map is contained within a secret area.

Secret areas are created during NetHack map generation by marking certain traversable spots of the map as hidden. Both corridors as well as doors (areas that transition between rooms and corridors) can be marked as hidden (with a 1/8 chance for a door, and 1/100 chance for a corridor) [NetHack Dev Team15]. On average, there are seven hidden spots in a level. These hidden spots initially appear to the player as regular room walls (if generated as doors) or as empty spaces (if corridors) and cannot be traversed. The player can discover and make traversable a hidden spot by moving to an adjacent square and using the 'search' action, which consumes one turn. The player may have to search multiple times since revealing the secret position is stochastic.

Just like regular movement, searching consumes actions. The choice of locations to search, therefore, as well as the number of searches to perform at each location, must be optimized in order to preserve food resources. Intuitively, we would like to search walls adjacent to large, unexplored areas of the map for which there do not appear to be any neighbouring frontiers. Similarly, corridors that end in dead-ends are also likely candidates

for secret spots, as seen in Figure 2.2.

Figure 2.2 A NetHack map with the player having visited all non-secret positions. The vast majority of the map is still undiscovered, likely due to the presence of a secret corridor immediately north of the player's current position.

Due to the random positioning of secret doors and corridors in NetHack, it is not a good idea to attempt to discover every single hidden spot on a map. Some secret doors or corridors may lead to nowhere at all, or perhaps lead to another secret door which opens into a room that the player has already visited. Depending on map configuration, a player may be able to easily spot such an occurrence and avoid wasting time searching in those areas, but in other cases it might not be so easy. There is also a tradeoff between finding all disconnected rooms in a map and conserving turns; if only a small area of the map seems to contain a hidden area, then spending a large effort trying to find it may not be worthwhile.

2.1.2 Combat in NetHack

Combat is another essential part of NetHack, and one that lends perhaps the most difficulty to the game. The vast majority of player deaths can be attributed to dying in battle to one

of the over 375 monsters lurking in the dungeons, deaths caused either by incorrect combat movement tactics, items used, or unfortunate choice of equipment. In addition to the many monsters and their special properties and abilities, complexity also arises from the plethora of items the player can find and the choice of which to equip or use against any particular monster.

Each monster can have special attributes: some have brute strength, others can use magical items, and still others have intrinsic abilities such as disintegrating breath attacks, gaze of the Medusa, or splitting in two ad infinitum upon being attacked. A player must develop strategies for each particular monster in order to survive. These strategies commonly involve the choice of a specific weapon to attack with, armor to equip, and/or the use of special scrolls, potions, wands, spells, and other miscellaneous items. As an example, a player must be careful in choosing a weapon to use against a pudding-type monster, since contact with iron would cause the pudding to divide in two [NetHack Wiki17a].

Monsters are randomly placed throughout the levels of the game based on the dungeon level number; more difficult monsters appear further on in the game. Other monsters are not generated randomly, appearing (possibly only once) on a fixed level. Each monster has an experience level and difficulty rating. Experience level influences toughness in combat while difficulty rating takes into account damage output as well as special abilities. Toughness is also affected by dungeon level and player experience level.

Items are also randomly placed throughout the dungeon. As mentioned above, there are many different item types: weapons (melee and ranged), armor (helms, gloves, etc.), scrolls, potions, wands, and more. Each item can be one of blessed, uncursed or cursed (referred to as BUC status); blessed items are more powerful than their uncursed counterparts while cursed items may cause deleterious effects. Further, each item is made of a certain material (iron, silver, wood, etc.); materials interact with the properties of certain monsters (e.g., many demons are particularly susceptible to silver weapons). Weapons and armor can also have an enchantment level (positive or negative) which relates to their damage output, as well as an integrity level ('condition') based on their material (wood items can be burnt, iron items corroded, etc.).

Resource management, mentioned earlier with respect to movement and food, also comes into play here. Due to the scarcity of items, those with more powerful effects should

be conserved for stronger monsters and not be wasted on the weak; we do not deal with this issue in this work.

2.1.3 Interfacing with NetHack

Throughout the past decades there have been quite a few automated players ('bots') created for NetHack, and all face the same problem of speed. Since NetHack is a game played through a console window, and sometimes hosted on servers, there is an inherent delay in sending and receiving game commands and state information. In addition to this delay, there are other associated difficulties such as emulating a terminal interface, managing a Telnet connection, etc. The delay in particular would make any large-scale experiments, such as those involving reinforcement learning applications, intractable in this context.

Several different approaches have been proposed to solve this issue over the years [MLO+09], but here we introduce a new method. We directly modify the NetHack game code in order to support the creation of two-way socket communication (using the *ZMQ* library [Hin11]) between the game and experiment processes. Through sockets we can send the keyboard character corresponding to the chosen action we want to take, and NetHack will send back the currently-visible game map and player attributes/statistics, i.e., all information otherwise visible to a player via the typical terminal interface. The use of sockets in this manner dramatically improves the speed of game playthroughs and eliminates issues relating to interception of a terminal interface. This approach is similar to what was done for the game of Rogue with the ROG-O-MATIC bot [MJAH84].

2.2 Occupancy maps

A popular algorithm for exploration purposes sometimes used in robotics is known as occupancy mapping [ME85, Mor88]. This approach, used in conjunction with a mobile robot and planning algorithm, maps out an initially unknown space by maintaining a grid of cells over the space, with each cell representing the probability that the corresponding area is occupied (by an obstacle or wall, e.g.). As the robot moves around and gathers data about the environment from its various sensors, said data is integrated into the occupancy map.

With this data structure, knowledge within a certain confidence margin can be established as to which areas of the space are traversable, with the data from different sensors being combined to even out sensor inaccuracies.

A *frontier* is a term often used in conjunction with occupancy maps and exploration. It is a discrete point (coordinate) in an occupancy map on the boundary between known and unknown space. For example, a known door in a NetHack map which has not yet been visited would be considered a frontier. Planning in occupancy maps, i.e., choosing where to move next, is performed by visitation of selected frontiers, with selection often occurring through use of a frontier evaluation function.

Research using occupancy maps is usually intertwined with these sensor concerns, and so is not relevant to the current video game environment. However, what is retained is the idea of mapping a space onto a discrete grid with probabilities for obstacles. This same basic idea of occupancy maps has also been incorporated into a similar algorithm in video game research, detailed below. There is also other work in robotics dealing with probabilistic target search that does not involve occupancy maps, such as the Bayesian approach by Bourgault et al. [BFDW06].

2.2.1 Occupancy maps in video games

A flavour of occupancy maps has also been used in video games, proposed by Damián Isla. This algorithm is significantly different than the original robotics conception and is geared towards searching for a moving target in a video game context [Isl05]. It has been used in at least one game to date [Isl13].

Like the original occupancy map, in this algorithm a discrete grid of probabilities is maintained over a space (e.g., game map), but here a probability represents confidence in the corresponding area containing the target or not. A non-player character (NPC) can then use said map to determine where best to move in order to locate a target (such as the player).

At each timestep, after the NPC moves, probabilities in the grid will update to account for the target's movement in that timestep. At any timestep, the searcher (NPC) can only be completely confident that the target is not in the cells within its field-of-view, and so

can set those cells to 0 probability. If the target is in sight, then the NPC can simply move towards them; if not, then probabilities in the grid will diffuse to their neighbours, to account for possible target movements in the areas outside the searcher's current field-of-view. Diffusion for each cell n outside of the NPC's field of view at time t is performed as follows:

$P_{t+1}(n) = (1-\lambda) P_t(n) + \frac{\lambda}{|neighbours(n)|} \sum_{n' \in neighbours(n)} P_t(n')$

where $\lambda \in [0,1]$ controls the amount of diffusion.

Our implementation of occupancy maps borrows concepts from Isla's formulation, namely the idea of diffusion, which is repurposed for an exploration context.

2.3 Reinforcement learning

Reinforcement learning (RL) is a technique that allows an agent to explore the state space of an environment with the goal of maximizing reward. By taking actions, which alter the current environment state, and receiving rewards, the agent can learn the best action to take in any state. We say that an agent observes a state $s_t \in S$ at each timestep t, chooses an action $a_t \in A$ to perform in that state, and then observes a reward R_{t+1} and resultant next state s_{t+1}. The goal of the agent will be to maximize its long-term reward over the course of its actions in the environment.

The formalization of the classical RL environment comes in the form of a Markov Decision Process (MDP). An MDP consists of a set of states and actions, as well as transition probabilities from each state to another given any action, and a reward function that outputs a numeric value for each state-action pair. As given by the name 'Markov,' states in the MDP must have the Markov property: each state must comprise sufficient information of the past history of observed states; i.e., the probability of observing a reward r_{t+1} and next state s_{t+1} can be conditioned solely on the current state s_t and action a_t, and not all prior states $s_0 ... s_{t-1}$ and prior actions $a_0 ... a_{t-1}$.

An RL agent in an MDP will learn a policy π, which maps states to action probabilities. $\pi(a|s)$ refers to the probability that action a will be chosen in state s. The optimal policy in an MDP, π_*, seeks to maximize the sum of obtained rewards or *return*, i.e. $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$ The γ parameter here is referred to as the discount factor, and controls

how much the agent favors rewards farther into the future versus immediate rewards.

While following a policy π, we can define the value of a state-action pair, $q_\pi(s,a)$. In any particular state s while following π, an agent wishing to maximize optimal return can choose the action $a \in A$ that maximizes $q_\pi(s,a)$. The optimal policy π_* is greedy with respect to the action-values for each policy: $q_*(s,a) = max_\pi q_\pi(s,a)$.

An important topic in learning algorithms is the problem of exploration versus exploitation. The goal of an agent is to exploit the environment, by choosing the best action in each state to maximize reward (i.e., following the best policy). However, in order to determine the best action for any state, the environment must first be explored: only by trying all actions in each state can the agent establish which is best. A common way to let the agent explore is to introduce forced exploration at the policy level. One technique, known as ε-greedy exploration, lets the agent take a non-optimal (exploratory) action with probability ε, and otherwise take the current optimal action. ε is often annealed throughout training from a large to small value, in order to do more exploration at the beginning when action-values are uncertain, and less at the end when they have been tried more often.

More details on reinforcement learning, as well as Q-learning and the other topics discussed later in this section, can be found in the comprehensive reinforcement learning textbook by Sutton & Barto [SB98].

2.3.1 Q-learning

One algorithm used to find an approximation to the optimal action-value function, q_*, and thus approximately optimal action choices for each state, is called Q-learning. Q-learning updates action-values using the following formula, which takes into account the current action-value for state s and action a (i.e., $q(S_t,A_t)$), the reward observed by taking action a in state s (R_{t+1}), and the value of the best action in the next observed state [WD92].

$$q(S_t,A_t) = q(S_t,A_t) + \alpha[R_{t+1} + \gamma max_a q(S_{t+1},a) - q(S_t,A_t)]$$

where α is a parameter that influences learning rate and γ is the aforementioned discount factor. Convergence of the algorithm requires that all state-action pairs be visited in the

limit infinitely often.

Q-learning is an *off-policy* algorithm. Off-policy algorithms are those that employ two policies: the target policy π, which is what we seek to optimize as usual, and the behaviour policy b, which is what we use to generate data with which to update the target policy. For example, in Q-learning, we could have the behaviour policy be an ε-greedy policy, and the target policy be a fully greedy policy (i.e., always picking actions with respect to the highest action-value). Off-policy learning is useful in several cases and in some necessary, e.g., when we have been given data generated by some unknown, non-optimal method (not necessarily from a learning algorithm) and seek to derive from it an optimal policy; in such a case, the unknown data can be thought of as coming from the behaviour policy, with optimal as greedy policy. It is also useful since it allows for more exploration while still learning the optimal policy. However, off-policy algorithms also present various stability issues since they introduce variance (caused by the differences in the two policies) and may be slower to converge than on-policy methods.

Another issue with Q-learning is scalability. In regular Q-learning, action-values are typically stored in a tabular form, with each table row having a state and its associated action-value (or alternatively, a dictionary-type data structure). As the algorithm progresses, empty action-values get filled in and others updated. This situation becomes less tractable as the state space grows; more and more memory must be devoted to the table's maintenance. At a certain point, the memory requirements become too onerous. In a large enough state space, it is unlikely that the same state may even be observed more than once, leading to low accuracy anyway. These problems motivate the need for a more efficient method to keep track of action-values, which brings us to the topic of function approximation.

2.3.2 Learning with function approximation

In situations where a state space is intractably large, the tabular setting is dispensed with to make way for the use of a function approximator. Such an approximator can be thought of a black box that takes in a state, does some sort of processing, and returns an action-value. The regular tabular action-value form is subsumed under this, but a table always produces

correct results – that is, the value you record in it for a state is the same value you get out if you query that state once more. A function approximator is less transparent: a change to the action-value of one state may at the same time modify in some unknown way the action-values of other states. This perhaps alarming property is actually quite useful since it gives the concept of generalization.

As mentioned above, some state spaces are so large that even with very long training times, a state may never be visited more than once or twice. Thus it becomes futile in these settings to run an RL algorithm that stores action-values in a table (although some approaches do exist), since without infinite visits to each state, an optimal or even near-optimal policy cannot be found. However, with function approximation, this restriction can be lifted. Learning with function approximation allows for experience with a small subset of the large state space to generalize over a much larger subset with reasonable accuracy. The key idea is that modifying the action-value of one state will also modify the action-values of 'related' states. Thus by visiting one state, we can actually gain information about a much larger set of states.

Such generalization requires a way to infer which states are similar to other states, in order for them to be 'grouped together' in some fashion so that an update to one will update to some extent the others in the group. Luckily, other fields of machine learning, i.e., supervised learning, are studying this very concept. Methods such as artificial neural networks are prime contenders for the aforementioned function approximator; they will be discussed here as they are most relevant to the presented work.

2.3.2.1 Neural networks

Neural networks take in input (e.g., the state) and return an output (e.g., the action-value), and through the process of supervised learning, strive to approximate a function from the data, with this approximation giving generalization of input features. There is quite a bit of background needed for a proper understanding of neural networks; here we give a brief overview with a focus on parameter settings, since they will be discussed further on.

A neural network is made up of *layers*, with one input layer, one output layer, and possibly one or more hidden layers in between, as shown in Figure 2.3. Each layer is made up of

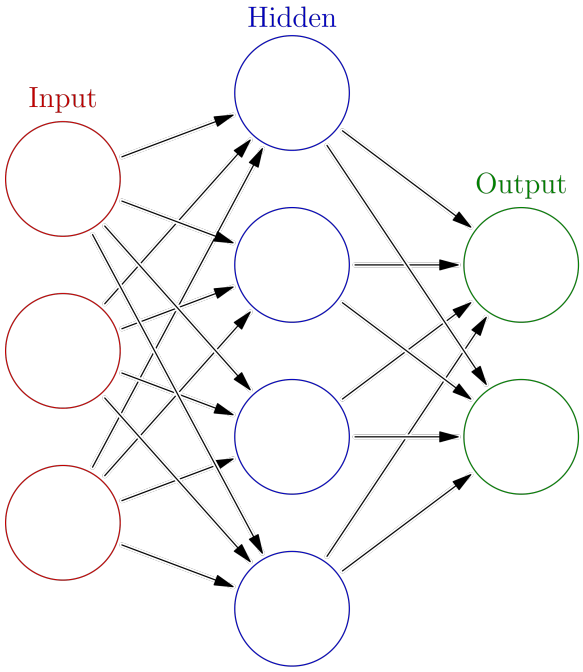

Figure 2.3 An example of a fully-connected, feed-forward neural network with an input layer of 3 units, one hidden layer of 4 units, and an output layer of 2 units [Glo13] (licensed under CC BY-SA 3.0).

a configurable amount of *units*. In a regular, *fully-connected*, *feed-forward* neural network, the units of each layer are connected to all units of the following layer. Each connection, or *link*, between units has a particular weight associated with it. These weights "roughly correspond to the efficacy of a synaptic connection in a real neural network" [SB98]. Input to the neural network will travel from the input layer to each successive layer on all links; each unit, on receiving input, will "compute a weighted sum of their input signals and then apply to the result a nonlinear function [...] to produce the unit's output" [SB98]. An example of an activation function is the logistic function.

The more hidden layers in the network, the better the function approximation. Even with one hidden layer, a vast number of continuous functions can be learned, and most applications seem in practice to not need more than a few. Recently, however, the use of multiple hidden layers has come into vogue with the popularity of convolutional neural networks, which are specially designed to deal with image or time-series data. However,

non-linear neural networks (that is, networks with one or more hidden layers) are less well understood theoretically and have less guarantees compared with the standard linear approaches.

A neural network is usually trained using the backpropagation algorithm. During training, the weights attached to each link between units are updated in order to minimize loss and thus increase accuracy of the network on future training iterations. Through training, units can 'specialize' in certain parts of the input, increasing their weights on certain inputs and decreasing on others; it is sometimes said that units pick out different 'features' of the data. When using a neural network as replacement for the action-value table in RL, the neural network's task is to approximate the optimal action-value function, Q_*.

The use of neural networks as function approximators has important advantages: they allow generalization amongst similar states, and their weights have very low memory requirements compared to tabular algorithms. But their use also presents some difficulties. One issue that frequently arises in supervised learning and is present to some degree in RL is that of overfitting. With certain combinations of hyperparameters, or with limited training data, the neural network weights may obtain great success on training data, but fail when later presented with slightly different data, since the network's weights have become too specialized on the training data.

When combining non-linear function approximation with off-policy learning (such as Q-learning), however, larger issues like action-value instability or divergence begin to manifest; see Baird's counterexample for an in-depth explanation [Bai95]. To come up with a solution that works at least in application to these problems, we turn to deep Q-learning.

2.3.2.2 Deep Q-Learning

Deep Q-learning (DQL) is a novel variant of the regular Q-learning algorithm, intended to correct instabilities that appear when using non-linear function approximation. The two main proposals of DQL, detailed below, are *experience replay* as well as introducing a second Q-network to store intermediary values, along with various other modifications. Introduced by the Google DeepMind team in 2014, this algorithm was shown to outperform human experts on several Atari 2600 arcade games when using a particular neural

network structure, including the use of convolutional neural networks, called a Deep Q-Network [MKS+13, MKS+15]. We use a similar approach for our combat agent but without convolutional neural networks, described in Chapter 4.

Experience replay is a technique that softens instabilities during the learning process by randomizing the order of data fed to the neural network. If data passed to the neural network is correlated in some way (as it would be if passed in sequence as the RL agent explores the environment), then the 'i.i.d.' assumption of the data (independent and identically distributed) is broken, leading to potential divergence. The solution is to create a buffer of past observations, called *experiences* or 'memories.' At each timestep, the agent's current state, chosen action, reward observed, and next state (s_t, a_t, r_t, s_{t+1}) are stored in the replay buffer. During learning at each timestep, experiences are sampled randomly from this buffer and used to update the neural network.

Another modification to soften instability is the introduction of a second neural network to generate targets for the Q-learning algorithm (for use in loss computation). On a certain fixed schedule, weights from the regular neural network are copied over to the target neural network. This delay dampens issues arising from large magnitudes in updates that could otherwise lead to large oscillations or divergence.

Other, smaller tweaks were also proposed, such as clipping rewards in the $[-1, +1]$ range, and the algorithm was further refined in [HGS16] and [WSH+16].

Chapter 3
Map Exploration

In this section we detail the basic exploration algorithm involving occupancy maps, and contrast it with a simpler, greedy approach as well as an approximately optimal solution. Key to our algorithm is the idea of limiting exploration to a subset of interesting space in order to minimize exploration time, by taking into account frontier utility and distance. We begin by discussing the modified NetHack environment in which the algorithms will run, followed by an outline of each algorithm with and without support for detecting secret areas. Results and discussion close the chapter with an emphasis on analysis of algorithm parameters.

3.1 NetHack environment

A modified version of the base NetHack game is used to test our exploration algorithms. Mechanics that might alter experiment results were removed, including monsters, starvation, weight limitations, locked doors, and certain dungeon features that introduce an irregular field of view. In addition, a switch to enable or disable generation of secret doors and corridors was added. Some of these limitations are relaxed in a later chapter.

The maps used in testing are those generated by NetHack for the first level of the game. An example of a fully revealed map can be seen in Figure 3.1. The same level generation algorithm is used throughout a large part of the game, so using maps from only the first level does not limit generality. Later levels can contain special, fixed structures, but there is no

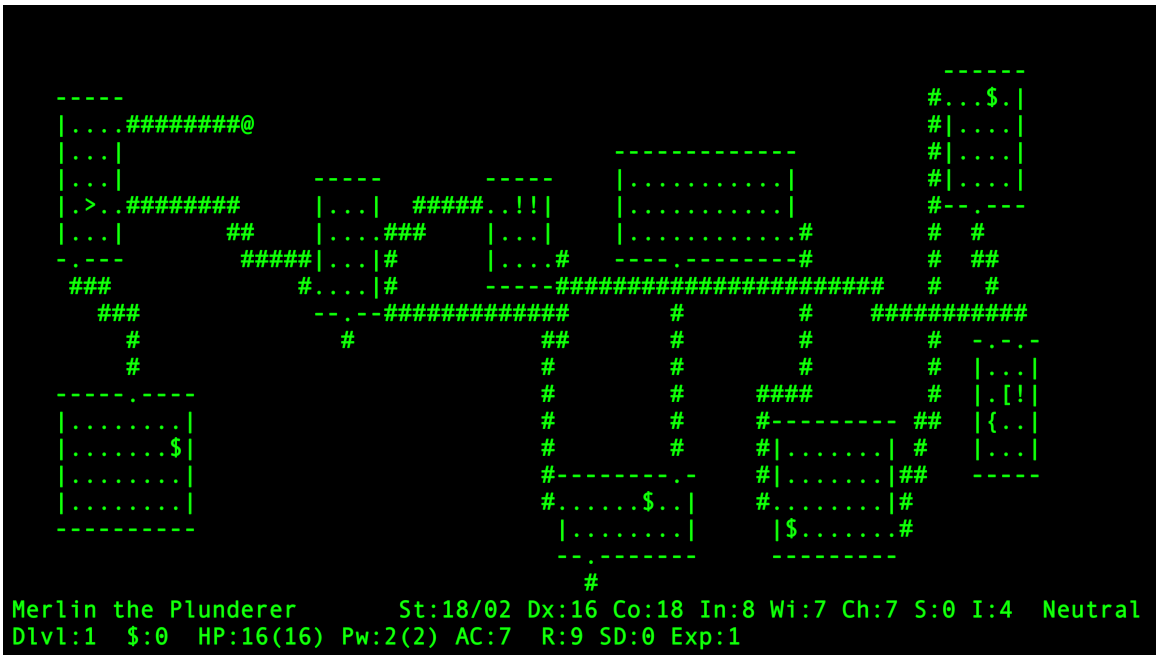

Figure 3.1 An example of a fully-explored NetHack map. In this map, a large corridor in the middle connects many rooms together.

inherent obstacle to running our algorithm on these structures; we are just mainly interested in applying exploration to the general level design (basic room/corridor structure).

The algorithms below use the standard NetHack player field of view. When a player enters a room in NetHack, they are able to immediately perceive the entire room shape, size, and exits (doors). Conversely, in corridors, knowledge is revealed only about the immediate neighbours to the player's position. Our algorithms will gain the same information as the player in these cases. We do not however support 'peeking' into rooms, where a player can perceive a portion of a room by being parallel to and within a certain distance of one of its doors.

3.2 Exploration algorithms

Here we present three algorithms: a trivial greedy approach which guarantees complete coverage of a space, an approximately optimal room visitation algorithm (given full prior knowledge of the map), as well as a nuanced approach based on occupancy maps, which

will only visit frontiers considered useful and do so in a specific order.

3.2.1 Greedy algorithm

A greedy algorithm is used as baseline for our experiments, which simply always moves to the frontier closest to the player. This type of approach is often formalized as a graph exploration problem, where we start at a vertex v, learn the vertices adjacent to v, move to the closest unvisited vertex (using the shortest path) and repeat [KTH01]. The algorithm terminates when no frontiers are left. We also take into account the particularities of the NetHack field of view as described above; when we enter a room, all positions in the room are set to visited, and its exits are added to the frontier list.

Note that this formulation will by nature uncover every traversable space on the map, both rooms and corridors alike.

3.2.2 Approximately optimal algorithm

For a lower bound on the number of moves needed to visit all rooms of a NetHack map, we present an approximately optimal algorithm. We call the algorithm 'optimal' since it will be given the full map at the start and so can plan the best route to take for room visitation. It is only approximate since it will seek to visit the centre of each room, while a faster version could move from room exit to room exit, avoiding the centre and thus saving a couple of moves per each room on a map.

To run this algorithm, we construct a complete graph where each vertex represents the centroid of a room on the current NetHack map, and edges between room centroids represent the shortest distance between them in map units (calculated using A*). We then pass this graph to a travelling salesman problem solver, along with the player's starting room. In order to prevent the TSP solver from returning to initial centroid at end, we add two 'dummy' vertices, one with a connection to every other vertex at cost of 0, and the other connected to the starting room and other dummy vertex with cost of 0, as suggested by [LLKS86].

This solution will guarantee exploration of all rooms, but not necessarily all corridors (similar to the occupancy map algorithm, below). It is thus a lower-bound to said algo-

rithm, but unlike the former cannot explore intrinsically since it must know the full map in advance.

Note that this problem is similar to the shortest Hamiltonian path problem, which attempts to find a path that visits each vertex on a map, but requires that each vertex only be visited once which could sometimes lead to suboptimal routes and in other cases fail to find a solution (e.g., if the graph has more than two terminal vertices).

3.2.3 Occupancy map algorithm

An exploration strategy typically has two key parts: the internal representation of the space to be explored, and how said representation is used in deciding where to move next. Both components of our strategy will be described below, in addition to a detailed look at how diffusion, a concept from Damián Isla's algorithm for searching for a moving target, is used as the engine that drives planning. Diffusion will also help in deciding when to stop exploring, another important component.

The main goal of the algorithm is to optimize exploration time by prioritizing visitation of areas most likely to confer benefit (rooms) while minimizing travel in unhelpful areas (corridors). To identify which unexplored areas are more likely to confer benefit, we combine an occupancy map as representation with a frontier list and frontier evaluation function. As mentioned earlier, only the rooms (and not corridors) of a NetHack level contain food (necessary for survival) and other useful items, so minimizing corridor visitation (by ignoring certain frontiers) does not have any drawback with regard to food/item collection.

A key parameter of the algorithm is the probability threshold value. The threshold value controls in a general sense the cutoff for exploration in areas of lower benefit; a higher value will mark more frontiers as unhelpful and thus focus exploration on areas of higher potential benefit (giving a tradeoff between time and amount explored). This threshold can be fixed at the start of the algorithm, or in another formulation, vary depending on the percentage of map uncovered (ignoring more frontiers as more of the map gets uncovered).

3.2.3.1 Representation

To represent the map of a NetHack level we use a structure similar to an occupancy map, which will store information about the map as in robotics. However, there are certain key differences here, since we would like to have a data structure that helps us determine general areas that are beneficial to visit (i.e., locations of as yet undiscovered rooms in a NetHack map), not just locations of obstacles (walls).

In robotics, an occupancy map is used to mark areas that contain obstacles; here we use it to mark visited (open) areas. Each cell of our occupancy map contains a probability, like in robotics, but instead of representing the likelihood of an obstacle, here it is rather an estimate of how likely that cell/area is to contain an unexplored room. Thus, a probability of zero means there is no chance an unexplored room can be found in that cell; we thus assign zero probability to any already visited room/corridor cell. Specifically, whenever we observe a room/corridor, we add its coordinate(s) to our memory; at each timestep, we set the probability of each coordinate's cell in our memory to 0 in the occupancy map (they are reset to 0 at each timestep since the diffusion step below may alter them). After setting a cell to 0 for the first time, all cells in the grid are then re-normalized to ensure the total map probability sums to 1.

Figure 3.2 gives a visualization of a sample occupancy map, with darker areas corresponding to lower probabilities (e.g., visited rooms).

3.2.3.2 Diffusion

Diffusion of probabilities is a central concept in Isla's algorithm, as described in Chapter 2.2.1, and we here adapt it for two purposes: to elicit a gradient of probability that flows from visited areas into unknown areas, in order to better measure the utility of frontiers, as well as to separate the occupancy map into distinct components of high probability. We leave explanation of the latter purpose for a later section, here discussing the former, in addition to describing how and when to run diffusion.

Diffusion affects the utility of a frontier. By dispersing the zero probability of visited rooms into surrounding areas, frontiers close to low probability areas can more easily be identified and ignored during exploration. This effect is desirable since these frontiers

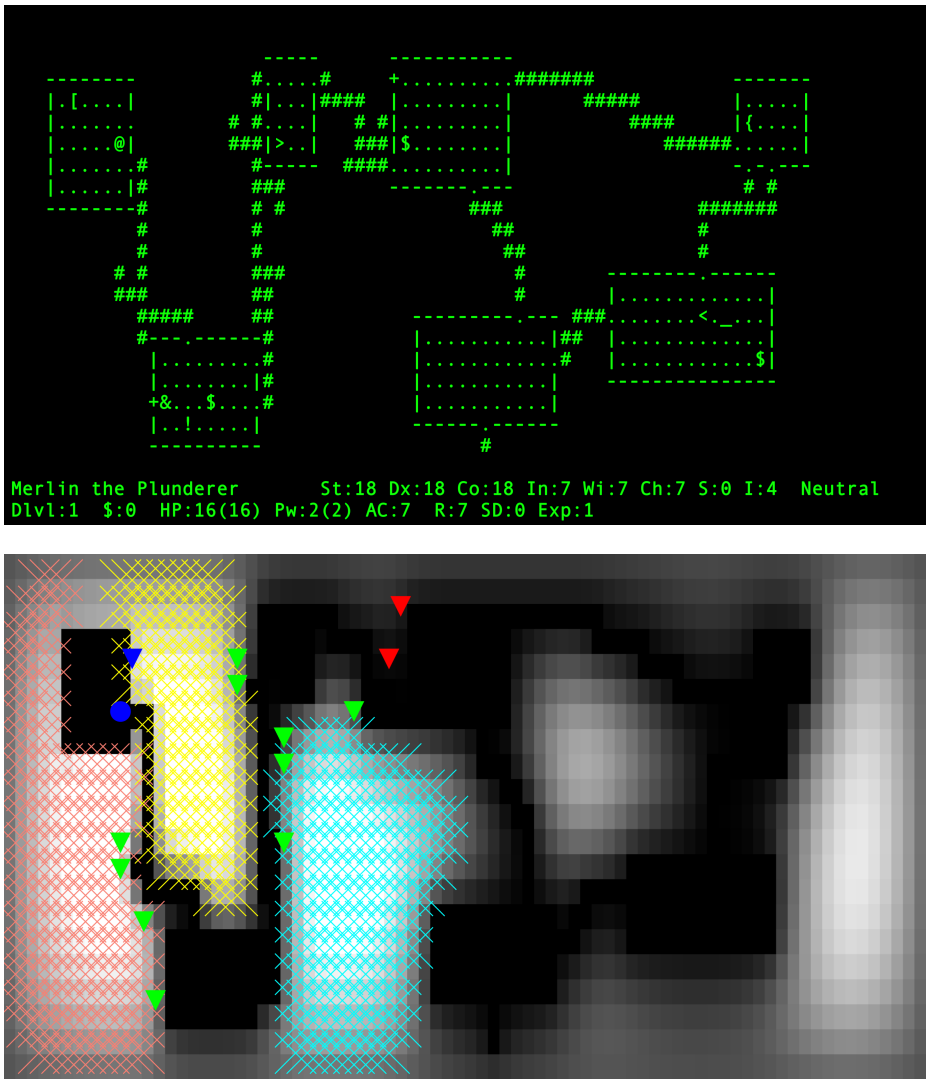

Figure 3.2 Visualization of an occupancy map (bottom) with corresponding NetHack map (top). Lighter areas are more likely to contain an undiscovered room. The player is shown as a blue circle, and current target frontier shown as blue triangle. Other frontiers are shown as green triangles, while red triangles represent frontiers that will not be visited due to being in areas of low probability. Components with neighbouring frontiers are highlighted in a criss-cross pattern, while components without nearby frontiers are not marked.

likely do not lead to as yet undiscovered rooms. Figure 3.2 shows these low utility frontiers as red triangles. In particular, a frontier is ignored when all of its neighbouring cells have probability below the threshold value. For a more forgiving measure, the neighbours of

neighbours could also be checked – or neighbours up to n distance away.

An example the advantageousness of this diffusion can be seen in the NetHack map of Figure 2.1. At the bottom of the map, there is a room in the centre that has an unopened door on its left wall. A few spaces past this wall, there is a visited corridor spot. When the occupancy map algorithm is run, the low probabilities from the visited corridor and visited room will diffuse towards each other, lowering the utility of the door/corridor frontiers. This behaviour is desirable since there is no need to visit what is highly likely to be a few more corridor spaces that only connect the two visited areas. The same effect can be seen in the map of Figure 3.2, where the centre-top room has an unopened door in its top-left corner; low probabilities from the nearby visited corridors have diffused towards the door and so it will not be visited.

Diffusion is performed by imparting each cell with a fragment of the probabilities of its neighbouring cells, as given in the diffusion formula in section 2.2.1. For extra diffusion, we also diffuse inward from the borders of the occupancy map. Specifically, when updating cells that lie on the borders, we treat their out-of-bounds neighbours as having a fixed low probability. Diffusing in this manner lessens the utility of frontiers that lie near the borders, which are in fact most likely dead-ends. It also tends to increase separation of components of high probability (since rooms/corridors rarely extend to the edge of the map).

Diffusion is only run when a new part of the map is observed (i.e., new room or corridor). By diffusing only at these times, probabilities in the occupancy map will not change while we are travelling to a frontier through explored space (and neither will the length of distance travelled have an effect). Probabilities will diffuse at the rate that map spaces are uncovered, and stop when the map is completely known.

This scheduling is the opposite of the diffusion in Isla's algorithm, which diffused when the search target was not observed to account for possible movements of the target in unseen space. In that formulation, previously visited areas could have their probabilities increase from zero when outside of the searcher's field of view, to account for the possibility that the target moved back into those areas. In our case, however, the 'targets' (unexplored rooms) are fixed.

Pseudocode for maintenance of the occupancy map with diffusion of probabilities is located in Algorithm 1. Note that the occupancy map cell probabilities are each initialized

to the same fractional value such that their sum equals to one.

One difficulty with diffusion is that it affects all cells on the map. With a small map size (as in NetHack) and finely tuned parameters, this 'global' type of diffusion is not an issue, but difficulties could start to crop up otherwise. For example, in a large map, although a player in the middle can choose to move in any direction, diffusion will still affect all cells, even those that may be quite distant from the player's current position. This effect seems misguided here: if too much time is spent on one side of the map, then the other side may end up having very low probabilities even if it has not yet been visited. In short, the spatial and temporal order in which a map is explored can arbitrarily and perhaps negatively affect probability values in the map. This issue does not pertain to NetHack maps due to their size and so is left to future work. One corrective measure may be to consider a 'local' type of diffusion, which only affects cells in the current area, or perhaps exchanging continuous diffusion for some sort of one-off immediate probability update.

3.2.3.3 Planning

We now use the knowledge in this representation to select the most promising frontier to visit, while (as previously stated) ignoring frontiers that lie in areas of low probability. To do so, we need a global view of the areas of high utility in the map, in the form of collections of adjacent cells of high probability, or *components*. There are two basic parts to the process: identifying these components and their associated frontiers, and then evaluating them to find the most useful one. First we describe reasons for dealing with components instead of frontiers directly.

At any given time, many frontiers can exist: unvisited doors in rooms, unvisited spots in corridors, etc. Since we want to visit frontiers that have the highest probability of leading to an unvisited room, the utility value of visiting a frontier should in some way correspond to the amount of adjacent cells of high probability in the occupancy map. We call these collections of frontier-adjacent cells *components*. Each component could have multiple adjacent frontiers that are perhaps right next to each other, or that border disparate sides of the component. To make computation easier and better elucidate differences in value between frontiers, we first determine these general components, evaluate them (based on

Algorithm 1 Diffusion in the occupancy map. (Run each time a new coordinate is visited.)

def visit_coordinate(coord)

Globals:

TOTAL_COORDS = 80 * 20 ▷ NetHack map size
occ_map = defaultdict(1/TOTAL_COORDS) ▷ initialize occ. map to default probs.
visited_coords = set()

Parameters:

diffusion_factor $\in$ [0, 1] ▷ how much to emphasize neighbour probs. in diffusion

Algorithm:

```
prob_culled = 0
if coord ∉ visited_coords then
    prob_culled += occ_map[coord]
    occ_map[coord] = 0                                  ▷ set current cell to zero probability
    visited_coords.add(coord)
if coord.in_room() then
    for room_coord in get_current_room_coords() - coord do
        prob_culled += occ_map[room_coord]
        occ_map[room_coord] = 0                         ▷ set all cells of current room to zero
        visited_coords.add(room_coord)
if prob_culled > 0 then                                 ▷ Only run diffusion when new coordinate visited.
    for unvisited_coord in occ_map - visited_coords do  ▷ Normalization of probs.
        occ_map[unvisited_coord] = occ_map[unvisited_coord] / (1 - prob_culled)
    cur_occ_map = occ_map.copy()
    for coord in occ_map do                             ▷ Diffusion step.
        neighbour_probs = get_neighbour_probs_for(coord, cur_occ_map)
        occ_map[coord] = ((1 - diffusion_factor)*cur_occ_map[coord]) + (diffusion_factor / len(neighbour_probs)) * sum(neighbour_probs)
    for coord in visited_coords do                      ▷ Make sure all visited coords have prob. 0
        occ_map[coord] = 0
```

utility value and distance to player), then pick the frontier closest to the best component, instead of dealing with frontiers directly.

Components are retrieved by running a depth-first search (DFS) on the occupancy map that traverses any cell with a probability value above the threshold. To further increase separation of components, we do not visit cells that have less than a certain number of traversable neighbours, which helps to deal with narrow alleys of high probability cells that could otherwise connect two disparate components.

Algorithm 2 Getting connected components using DFS.

def get_components(prob_threshold, frontiers)

Parameters:

min_neighbours ≥ 0 ▷ min. num. neighbours for cell to be traversed by DFS
min_room_size ≥ 0 ▷ required minimum component size

Algorithm:

components = $\varnothing$
for each cell in occupancy map **do**
    if cell $\geq$ prob_threshold **and** cell $\notin$ any component **then**
        component = dfs(cell, prob_threshold, *min_neighbours*) ▷ set of cells
        if component.has_nearby_frontiers() **and** $|\text{component}| \geq$ *min_room_size* **then**
            components.add(component)
return components

Some components are ignored due to small size or absence of neighbouring frontiers. If a component is smaller than the minimum size of a NetHack room, it is impossible for a room to be there. Likewise, if a component has no neighbouring frontiers, it cannot contain a room since there is no access point (unless secret doors/corridors are enabled, as discussed later). Pseudocode for finding the components in the occupancy map is shown in Algorithm 2.

The visualization of a sample occupancy map in Figure 3.2 gives an idea of this process, with three components highlighted using a criss-cross pattern in different colours. Each of the three are cut off from the others because the neighbouring rooms have diffused towards the edges of the map (and the border has diffused towards them). Meanwhile, the unmarked component in the upper-right is ignored since it has no neighbouring frontiers.

Algorithm 3 Choosing the best frontier, considering component distance and utility.

def get_best_frontier(frontiers, components, prob_threshold, player)

Parameters:

distance_multiplier $\in$ [0, 1] ▷ importance of distance in component evaluation

Algorithm:

```
1:  max_component_val = 0
2:  best_frontier = ∅
3:  for component in components do                          ▷ find the best component
4:      min_dist = +∞
5:      frontier_for_component = ∅
6:      sum_dists = 0                                       ▷ for distance normalization, line 16
7:      for frontier in frontiers do                        ▷ get the best frontier for this component
8:          closest_cell = component.get_closest_cell_to(frontier)
9:          frontier_distance = dist_astar(player, frontier) + dist_mh(frontier, closest_cell)
10:         sum_dists += frontier_distance
11:         if frontier_distance ≤ min_dist then
12:             min_dist = frontier_distance
13:             frontier_for_component = frontier
14:                         ▷ get value of this component, factoring in utility & distance
15:     comp_utility = component.sum_probs() / sum(c.sum_probs() for c : components)
16:     norm_comp_distance = min_dist / sum_dists
                    ▷ the [0, 1] normalization of utility is omitted for space purposes.
17:     component_val = comp_utility + distance_multiplier * (1 - norm_comp_distance)
18:     if component_val ≥ max_component_val then
19:         max_component_val = component_val
20:         best_frontier = frontier_for_component
    return best_frontier
```

The list of remaining components are then passed through an evaluation function to determine which best maximizes a combination of utility and distance values. Utility is calculated by summing the probabilities of all cells in the component. The sum is then normalized by dividing by the sum of all probabilities in the map. To determine distance to player, the component is first matched to the closest frontier on the open frontiers list, by calculating the Manhattan distance from each frontier to the closest cell in the component. Distance from component to player is then calculated as: $d(\mathit{frontier}, \mathit{player}) +$

$d(frontier, closest\_component\_cell)$ with the first half calculated using A*, and the second half using Manhattan distance (since that part of the path goes through unknown space). This distance is then normalized by dividing by the sum of the distances for all frontiers for the specific component under evaluation. With the normalized utility and distance values, we pick the component that maximizes $norm\_prob + \alpha * (1 - norm\_dist)$, where α controls the balance of the two criteria. Pseudocode for component evaluation is shown in Algorithm 3. Note that this will bias the algorithm to choose larger components, as they will have higher summed probability than smaller components.

Algorithm 4 Main planning loop of the approach.

def plan_next_step()

Parameters:

threshold_val ∈ [0, 1] ▷ prob. threshold value

vary_threshold ∈ [True, False] ▷ whether to vary prob. threshold

Algorithm:

```
1: while true do
2:     prob_threshold = calculate_threshold(threshold_val, vary_threshold)
3:     frontiers = get_useful_frontiers(prob_threshold, radius)
4:     components = get_components(prob_threshold, frontiers)
5:     if frontiers = ∅ or components = ∅ then
6:         stop
7:     best_frontier = get_best_frontier(frontiers, components, prob_threshold, player)
8:     player.move_to(best_frontier)                    ▷ send command to NetHack
9:     visit_coordinate(best_frontier)
```

Once the best component is determined, the algorithm moves to the frontier matched to that component. On arrival, it will learn new information about the game map, update the occupancy map, and run diffusion. Components will then be re-evaluated and a new frontier chosen. Exploration terminates when no interesting frontiers remain. Pseudocode for the main planning loop of the exploration approach is presented in Algorithm 4.

3.3 Algorithms with support for secret detection

We present below the adaptations necessary to enable searching for secret doors/corridors for both the greedy and occupancy map algorithms. Secret doors/corridors are map coordinates which cannot be reached by simple movement alone – they must be intentionally searched for at additional cost. It is thus crucial to selectively pick where to search for them to minimize time spent.

3.3.1 Greedy algorithm for secret rooms

A trivial adaptation can be made to the basic greedy algorithm in order to support searching for secret areas. When entering a room, before proceeding to the next frontier, each wall of the room is searched for secret doors for a certain number of turns. Searches are also performed in dead-end corridors. If a secret door/corridor is discovered, it is added to the frontier list as usual. Exploration ends when no frontiers or search targets remain.

For efficiency, searching for secret doors in a room is done by first choosing the unsearched wall closest to the player, then moving to a spot adjacent to the wall that also touches the most walls still needing to be searched (since searching can be performed diagonally).

Note that this approach is not capable of finding all secret corridors in a level, since they may (rarely) appear in regular (not dead-end) corridors. However, searching all corridors would be too strenuous to handle this rare occurrence. The below occupancy map approach also ignores these rare secret corridors.

3.3.2 Occupancy map algorithm for secret rooms

The occupancy map algorithm has a natural extension to support the discovery of secret door and corridor spots. In the original case, components of high probability in the occupancy map with no neighbouring frontiers would be ignored, but here, these components are precisely those that we would like to investigate for potential hidden rooms. Below we detail the adjustments necessary for this extension.

The first modification relates to the component evaluation function. Since these 'hidden' components have by definition no bordering frontiers, the distance from player to frontier and frontier to component used in the evaluation must be adjusted. Instead of using a frontier to calculate distance, we will choose a particular room wall or dead-end corridor adjacent to the hidden component, and calculate distance using that.

The selection of such a room wall or dead-end corridor for a hidden component requires its own evaluation function. This function will likewise consider both utility and distance. Utility is given by the number of searches already performed at that spot. Distance is taken as the length from the spot to the player plus the length from the spot to the closest component cell. Distance to player is calculated using A*, and distance to closest cell by Manhattan distance. Walls whose distance from the component exceed a certain maximum will be ignored. Both distance and search count are normalized, the former by dividing by the sum of distances for all walls, and the latter by dividing by the sum of search counts for all walls. We then pick the spot that minimizes $norm\_count + \sigma * norm\_dist$, where σ is the parameter that controls the balance of the two criteria. The value is minimized in order to penalize larger distance and higher search counts.

The selected wall/corridor spot is used in place of a frontier in component evaluation which proceeds as described earlier. If, after evaluation, a hidden component is selected, then we will move to the closest traversable spot adjacent to the component's associated wall/corridor spot. In case of ties in closest distance, the spot adjacent to the most walls will be chosen to break the tie, since searches performed at a position will search all adjacent spots (including diagonally).

When the player reaches the search location, the algorithm will use the search action for a certain number of turns (a parameterized value), before re-evaluating all components and potentially choosing a new search target or frontier to visit. If a secret door or corridor spot is discovered while searching, it is added to the open frontier list and its probability in the occupancy map is reset to the default value. Diffusion is then run throughout the map since new information has been revealed.

It is possible for a hidden component to not contain a secret area. Thus, if a wall or dead-end corridor remains unchanged after a certain number of searches (a parameterized value), it will no longer be considered as a viable search target.

Exploration terminates when no components are left, or only hidden components remain and none have viable search targets.

Figure 3.3 presents a visualization of a sample occupancy map with secret doors/corridors enabled and corresponding NetHack map. The component on the left side (marked with a grid pattern) has no neighbouring frontiers and is thus considered a hidden component; nearby walls that will be considered for searching during evaluation are marked with blue squares. In this experiment, a low minimum wall distance was used, preventing walls in the lower room from being selected for evaluation.

3.4 Results

Results will be shown below for the greedy and occupancy map algorithms as a function of the exhaustive nature of their searching, followed by results for the algorithms that can search for secret areas. We will look first at the metrics to be used for comparison of the algorithms.

3.4.1 Exploration metrics

To evaluate the presented exploration strategies, we use as metrics the average number of actions per game (which we seek to minimize) as well as average percentage of rooms explored, taken over a number of test runs on randomized NetHack maps. As will be seen below, the presented algorithms tend to do quite well on these metrics. Thus, in order to get a more fine-grained view of map exploration which penalizes partial exploration, we also use a third metric which we call the 'exhaustive exploration' metric. This metric represents the average percentage of runs that explored all rooms on a map (counting as zero runs that failed to find one or more rooms). We do not use amount of food collected as a metric since in these maps food is uniformly randomly distributed amongst rooms, and so is highly correlated with the percentage of rooms explored.

For algorithms that support detection of secret areas, two further metrics are used: the average percentage of secret doors and corridors found, and the average percentage of 'secret rooms' found. Neither of these metrics are ideal, however, and it is important to

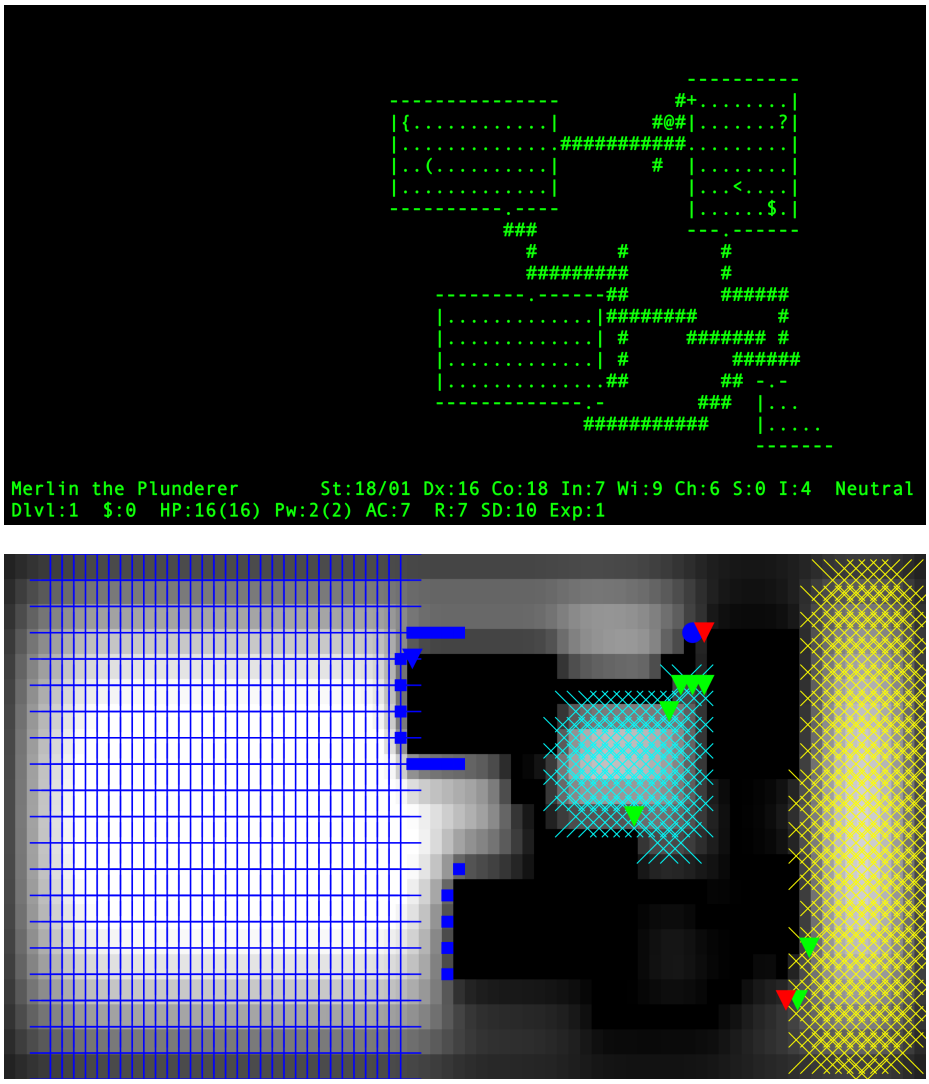

Figure 3.3 Visualization of a sample occupancy map (bottom) and corresponding NetHack map (top) with secret doors/corridors enabled. Hidden components are identified using a grid pattern while regular components use a criss-cross pattern. Highlighted squares near the hidden component represent the walls that satisfy the distance criteria.

understand limitations in evaluating secret room discovery.

The average percentage of secret doors/corridors found is problematic since it does not correlate well with actual benefit – only a handful of secret spots will lead to undiscovered rooms and so be worth searching for. Further, it is biased towards the greedy algorithm,

since that algorithm will search all walls, and so have a higher chance to discover more secret doors than the occupancy map algorithm, which will only search areas selected by its evaluation function.

The average percentage of 'secret rooms' found is also problematic, due to their ambiguous definition in the NetHack context. One possible way to define secret rooms here is to classify them as any room not directly reachable from the player's initial position in the level. In this case, the metric would be too dependent on the individual level configuration: a map could exist such that the player actually starts in a 'secret' room, separated from the rest of the map by a hidden door, and the algorithm would only have to find that spot to get a full score for this metric.

Further, while almost all maps tend to contain secret doors or corridors, only approximately half of all maps contain secret rooms as defined above (in the other half, any secret doors/corridors that do exist lead nowhere useful). This discrepancy also skews the secret room metric since maps containing no secret rooms will still get a full score using that metric.

3.4.2 Exhaustive approaches

Figure 3.4 presents results for the exhaustive exploration approaches (those that always explore all rooms on a map). Each result is an average over 200 runs on different randomly-generated NetHack maps. The greedy algorithm comes in at around 324 average actions per game, while the average for the fastest occupancy map model (with parameters that gave complete exploration on 99.5% of all runs) is 292 actions.

The greedy algorithm by nature explores all corridors, while the occupancy map algorithm limits exploration to areas likely to contain new rooms. The greedy algorithm is also a bit more reliable for complete room discovery than the occupancy map algorithm: we cited in the figure the occupancy map model that discovered all rooms in 99.5% of runs, meaning a small number of runs failed to discover all rooms on the map (missing one or two rooms in those cases).

In the same figure we present the result for the approximately optimal solution, which visits all rooms in 122 actions on average. This approach can only be applied to a fully-

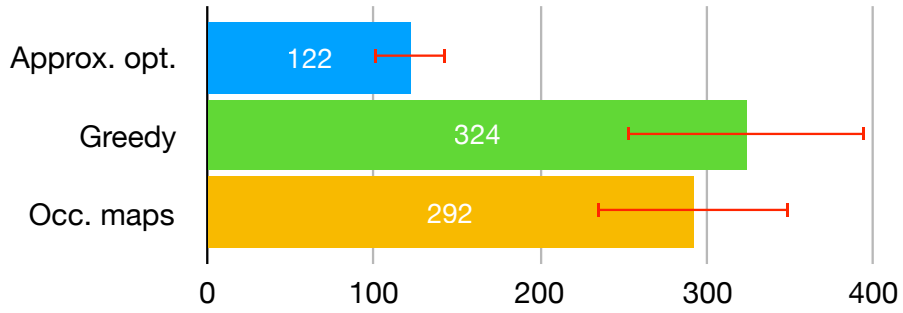

Figure 3.4 Average number of actions taken by the approximately optimal solution, greedy algorithm, and occupancy map algorithm for exhaustive room exploration with best performing parameters. The average over 200 runs on different randomly-generated NetHack maps is taken. Error bars (standard deviation over all runs) are presented in red.

known map and so does not lend itself to exploration, but is instructive as a lower-bound. The large discrepancy between this result and the other two algorithms is because this algorithm knows the room positions; the true exploration approaches can make mistakes in guessing.

Furthermore, the approximately optimal method knows to stop exploring when it has reached the final room. Other methods however may continue exploring unnecessary frontiers: the greedy approach will visit all frontiers and the occupancy map approach will only stop when confident that remaining frontiers are in areas of low utility. Figure 3.5 shows the number of actions taken by each approach on average until all rooms have been explored. The difference between these numbers and those in Figure 3.4 show how many actions on average are wasted by exploring useless frontiers near the end of each run: 58 actions for the greedy algorithm and 43 for the occupancy map.

3.4.3 Non-exhaustive approaches

Exhaustive approaches are desirable in certain circumstances, but it is often acceptable to occasionally leave one or two rooms on a map unexplored, especially when there is a cost to movement. Figure 3.6 gives the results for the best-performing non-exhaustive occupancy map models in terms of actions taken versus percentage of rooms explored. Each model

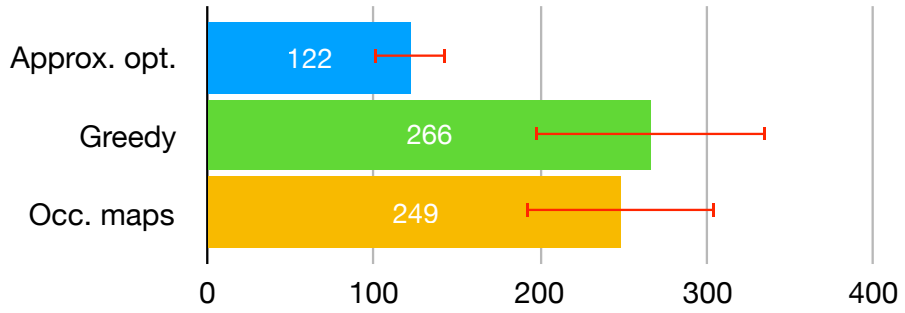

Figure 3.5 Average number of actions taken until all rooms are explored, by the approximately optimal solution, greedy algorithm, and occupancy map algorithm for exhaustive room exploration with best performing parameters. The average over 200 runs on different randomly-generated NetHack maps is taken. Error bars (standard deviation over all runs) are presented in red.

(represented by blue dots) represents an average over 200 runs using a unique combination of model parameters. The models shown lie on the upper-left curve of all models obtained by performing a grid search over the parameter space.

As seen in the figure, there is a mostly linear progression in terms of the two metrics. The relationship between the 'exhaustive exploration' metric and total percentage of explored rooms is also consistent, with both linearly increasing.

The figure shows that by sacrificing at most 10% of room discovery on average, the average number of actions taken can be decreased to 200, compared to the 292 average actions of the exhaustive (99.5%) approach or 324 actions of the greedy algorithm.

In general, the many parameters offer a fine-grained sweep of differences in the tradeoff between actions taken and amount of map explored. Figure 3.7 shows results for all 6,763 models, one model for each different combination of parameters.

3.4.3.1 Occupancy map parameter analysis

With such a large amount of parameters for the occupancy map algorithm, the question naturally arises as to which parameters really influence the results, and which can be for the most part left to default values. Therefore, in order to determine the importance of these parameters, a linear regression was performed. Parameter coefficients for average actions

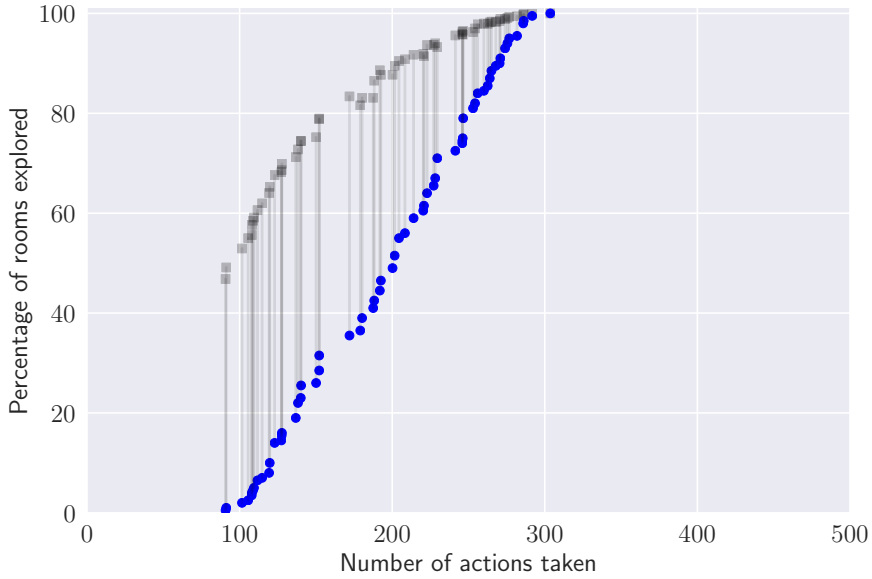

Figure 3.6 Occupancy map models with parameters that best minimize average actions per game and maximize percentage of rooms explored. Each blue dot represents the average over 200 runs using a different combination of model parameters. The blue dots show the result under the 'exhaustive exploration' metric and the corresponding black squares show the total percentage of rooms explored.

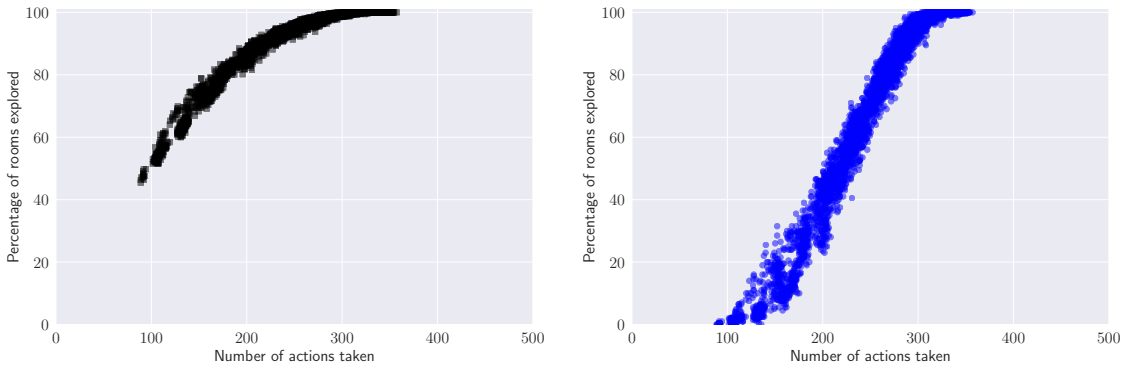

Figure 3.7 Occupancy map models, with each marker representing the average over 200 runs using a different combination of model parameters. Black squares (left) show total percentage of map explored while blue dots (right) show number of runs where all rooms were explored ('exhaustive exploration' metric).

and percentage of rooms explored under the 'exhaustive exploration' metric are shown in Figure 3.9. The specific ranges of values used are shown in Figure 3.8. R-squared values for the regression were 0.742/0.693 (for average actions and room exploration) on test data. Running a random forest regressor on the same data gave the same general importances for each parameter with more confident r-squared values of 0.993/0.993, but those importances are not presented here due to lack of indication of the correlation direction.

Parameter	Range
Diffusion factor	0, 0.25, 0.5, 0.75, 1
Distance importance	0, 0.25, 0.5, 0.75, 1
Border diffusion factor	0, 0.25, 0.5, 0.75, 1
Minimum room size	(3), 7
DFS min. neighbours	0, 4, 8
Prob. threshold	0, 0.15, 0.35, 0.5
Whether to vary threshold	False, True
Frontier radius	0, (1, 2)

Figure 3.8 Range of occupancy map algorithm parameters used in the grid search. Values in parentheses indicate these were not included in the complete grid search, since they did not seem to have an impact on results.

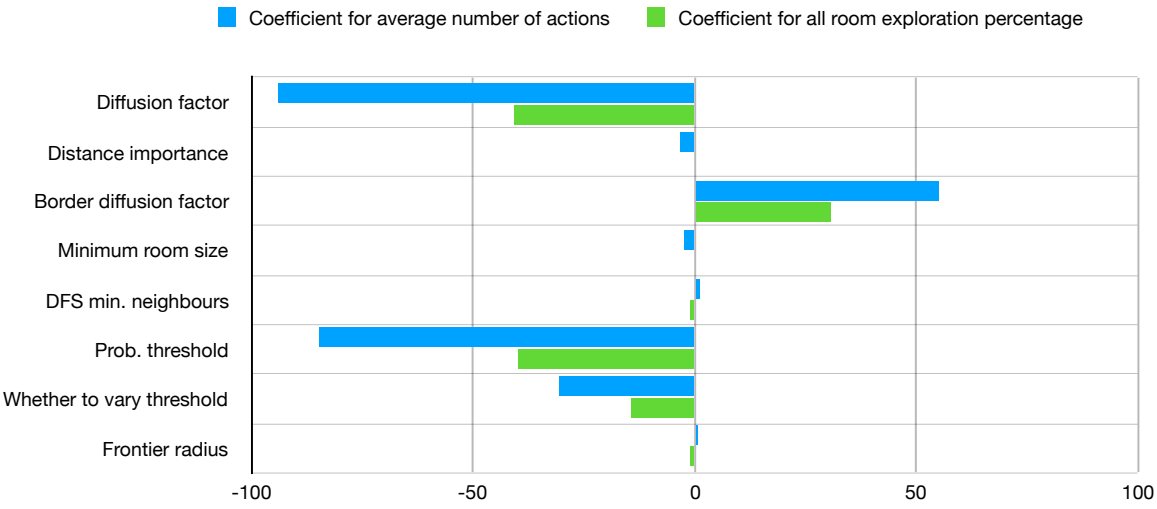

Figure 3.9 Linear regression coefficients for average number of actions and percentage of rooms explored with the occupancy map parameters as independent variables. Train/test data split is 70%/30% and dataset size is 6,763 (each datum being the result for a different combination of parameters).

The coefficients indicate that parameters directly associated with probabilities in the occupancy map are most influential on average actions and percentage of rooms explored. These parameters include the diffusion factor (how much to diffuse to neighbours), border diffusion factor (how much to diffuse from outer borders), probability threshold (at what probability to ignore frontiers, etc.), and whether to vary the threshold as more of the map is explored. The border diffusion is probably important due to the small (80x20) map size; on larger maps, it is less likely that this parameter would have such an impact.

Meanwhile, parameters that influence component size and choice, like distance factor (importance of distance in component evaluation) and minimum number of neighbours for a cell to be visited by DFS (which separates components connected by small alleys) did not seem to have a pronounced effect on the metric values. This finding may suggest that the location of frontiers, and ignoring ones that lie in areas of low probability, has more of an impact than the separation of components.

The specific parameter values that led to the fastest performing exhaustive exploration model (presented in Figure 3.4) were as follows: diffusion factor of 1, distance importance of 0.75, border diffusion of 0.75, minimum room size of 7, DFS min. neighbours of 4, probability threshold of 0.15, vary threshold set to false, and frontier radius of 0. The parameters for the fastest model at 80% non-exhaustive exploration (the full map being explored about 30% of the time) using 167 actions on average (as shown in Figure 3.6) were: diffusion factor of 0.75, distance importance of 0.25, border diffusion of 0.5 (smaller values diffuse more), minimum room size of 7, DFS min. neighbours of 8, probability threshold of 0.5, vary threshold set to false, and frontier radius of 0.

One parameter, the minimum component size, had very little effect on metric results. This parameter was introduced to decrease computation time. Small components would otherwise be eventually eliminated by diffusion anyway, and would be unlikely to be selected by the evaluation function due to their low summed probability, so the low effectiveness of this parameter is not surprising.

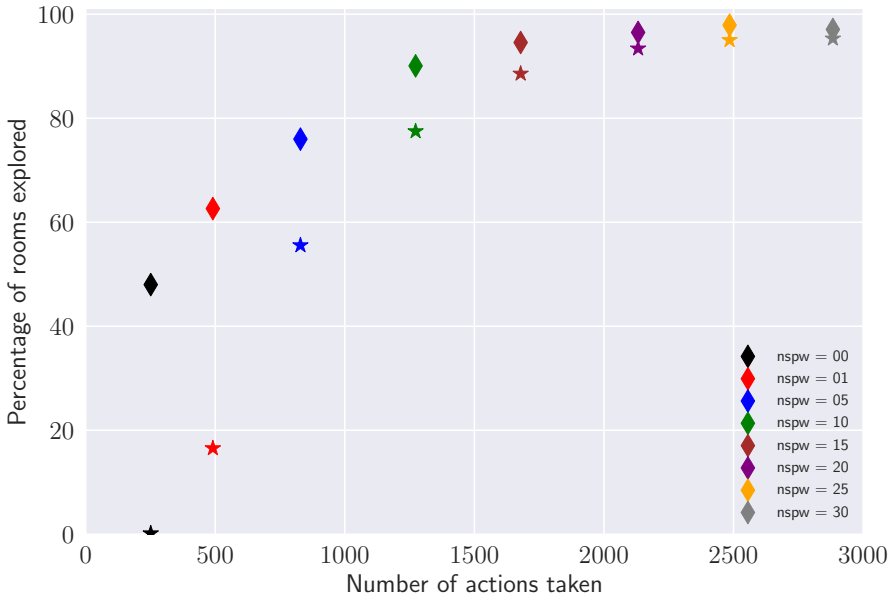

Figure 3.10 Number of actions taken by the greedy algorithm with support for secret door/-corridor detection with varying values for number of searches per wall parameter ('nspw'). The average percentage of secret doors/corridors found is represented by stars and average percentage of secret rooms represented by diamonds.

3.4.4 Approaches for secret rooms

Below we present and discuss results for the greedy and occupancy map algorithms with adaptations for searching for secret doors and corridors.

3.4.4.1 Greedy algorithm for secret rooms

Figure 3.10 shows the results for the greedy algorithm with support for secret detection in terms of average actions versus exploration. Different colours represent the different settings for the number of searches per wall parameter (the number of times the algorithm will search a particular wall/corridor before moving on). Both the average percentage of secret rooms found and average percentage of secret doors and corridors found are displayed.

As expected, both metrics increase as the number of searches per wall increases, plateauing at around 95% discovery of both secret rooms and secret doors/corridors at around 2250 average actions per game. As mentioned earlier, the algorithm will only search for secret

corridors in dead-ends, so it is missing about 5% of hidden spots probably because of the secret corridors occurring (rarely) in other locations.

Another observation is that when the number of searches per wall is set to 0, the algorithm is reduced to the regular greedy algorithm, with no secret doors/corridors being found (since there no searching is performed). The approximately 50% score for the secret rooms metric is due to the fact that, in that percentage of runs, there were no secret rooms at all, thus giving 100% exploration as mentioned in the metrics discussion.

3.4.4.2 Occupancy maps for secret rooms

Figure 3.11 gives the results for the best-performing secret-detecting occupancy map models in terms of best time versus highest secret room exploration. Each model represents an average over 200 runs using a unique combination of model parameters. The models shown lie on the upper-left curve of all models obtained by performing a grid search over the parameter space.

Results here are much better than the greedy algorithm, with approximately 90% secret room exploration at around 500 actions. The reason for the discrepancy between this result and the greedy algorithm (over 1600 actions for 90%) is that the occupancy map model has better, global knowledge of the map and can target particular walls for searching, in contrast to the greedy algorithm which searches every wall.

This global knowledge also explains the much lower percentage of secret doors/corridors discovered using this algorithm (20% for the model exploring 90% of secret rooms) compared to the greedy algorithm (80% for the model exploring 90%). This result is expected since exploration of secret doors/corridors only weakly correlates with secret room exploration (only a few secret doors/corridors will actually lead to otherwise inaccessible rooms).

Importances of the parameters for the secret-detecting occupancy map algorithm are shown in Figure 3.13. These importances were calculated by running a random forest regressor on the model results. R-squared value for the average actions coefficient was 0.864 on the test data, while for the secret room exploration coefficient, the value was much lower at 0.334, suggesting that some parameters are not linearly independent in relation to

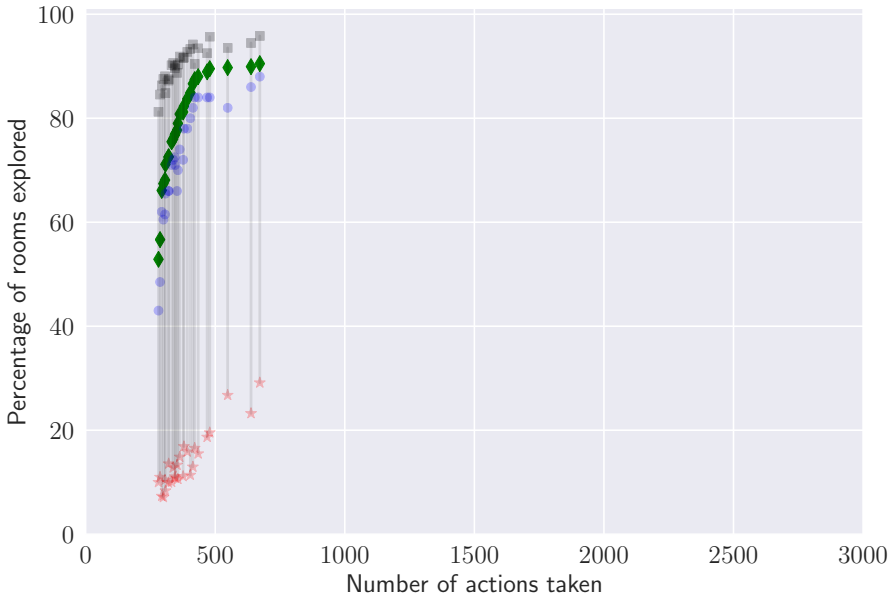

Figure 3.11 Occupancy map models with support for detecting secret doors/corridors, with parameters that best minimize average actions per game and maximize exploration of secret rooms. Green diamonds represent the percentage of secret rooms explored, while the corresponding red stars represent the percentage of secret doors/corridors explored. Also shown are the regular percentages of rooms explored, represented by blue circles and black squares as explained in the caption for Figure 3.6.

that variable.

The specific ranges of parameter values used in the grid search are shown in Figure 3.12. In order to save computation time, some parameter ranges are narrowed and some values fixed here to their corresponding best-performing values from the regular occupancy map algorithm. Figure 3.13 does not show importances for those parameters that have fixed values.

The importances show that the three diffusion-related parameters (diffusion factor, border diffusion and probability threshold) continue to have a large impact on the average actions and secret room exploration metrics. In addition, other factors that did not have any importance in the earlier occupancy map algorithm have a significant impact here, particularly the minimum neighbours for DFS. This parameter affects the separation of

Parameter	Range
Diffusion factor	0.25, 0.375, 0.5
Distance importance	0, 1, 2
Border diffusion factor	0.2, 0.6, 1
Minimum room size	**5**
DFS min. neighbours	2, 4, 6
Prob. threshold	0.2, 0.25, 0.3
Whether to vary threshold	False
Frontier radius	0, 2
Wall distance importance	0, 1, 2
Max. searches per wall	**7**
Minimum wall distance	5, 10, 15
Num. actions per search	**7**

Figure 3.12 Range of secret-detecting occupancy map algorithm parameters used in the grid search. Some values tested for partial ranges are omitted.

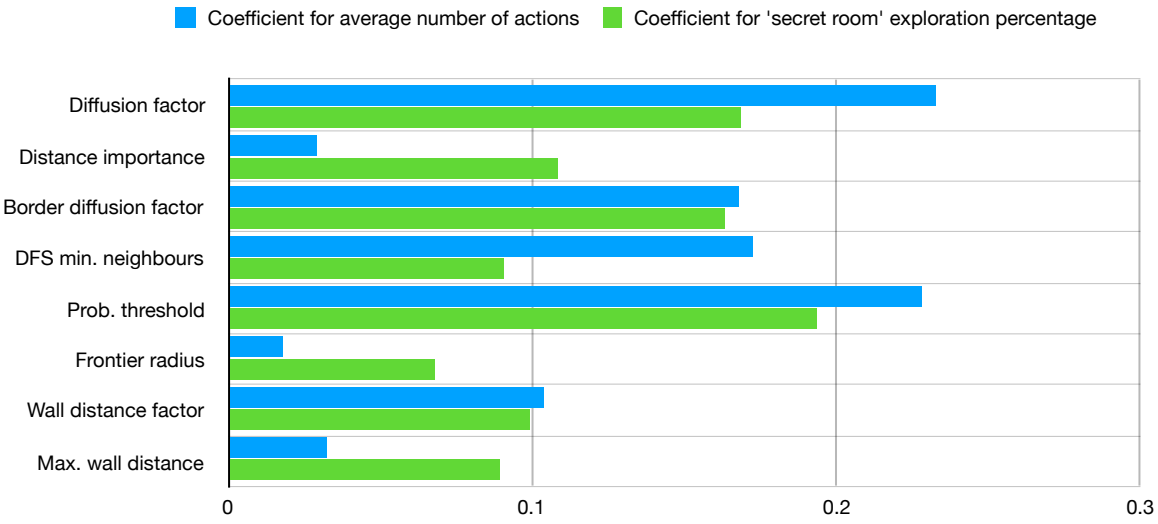

Figure 3.13 Random forest regression coefficients for average number of actions and percentage of secret rooms explored with the secret-detecting occupancy map parameters as independent variables. Train/test data split is 70%/30% and dataset size is 4,470 (each datum being the result for a different combination of parameters).

components, suggesting that the use of components for this algorithm matters more than in the earlier case.

Parameters exclusive to this algorithm also had somewhat of an effect on the dependent variables, including the wall distance factor (importance of distance in the choice of walls to search for a hidden component) and maximum wall distance (maximum distance between a wall and a hidden component before it is removed from consideration for searching).

3.5 Discussion

In this chapter we presented an algorithm to efficiently explore a NetHack map and contrasted it with a simpler, greedy approach. The algorithm involves the use of occupancy maps as well as the concept of diffusion from Damián Isla's work. To determine the best frontier to visit, the occupancy map is split into components of high probability and the most promising component to visit next is chosen according to utility and distance. We also detailed adaptations to this and to the greedy algorithm in order to support searching for secret areas, which cannot be found by traditional exploration alone. Finally, we presented results for the algorithms, and looked into the importances of the various parameters of the occupancy map algorithm which allow for different tradeoffs between amount of map explored and exploration time. This algorithm will be useful in contexts where time is a resource that must be carefully managed, like in NetHack and similar roguelikes. We now turn to another aspect of NetHack: combat.

Chapter 4
Learning Combat Strategies

In this section we describe our NetHack combat learning approach with custom rewards, states and actions. We also present a handful of simple baseline strategies and compare them with our approach. Combat is modeled in a one-on-one player-versus-monster arena setting, with a variety of equipment and items available to the player. This generic setting can naturally be extended to other roguelikes.

A learning approach to combat can be contrasted with existing rules-based approaches, which use hard-coded responses and strategies for each monster (as mentioned in Chapter 6). Learning these rules automatically alleviates the tedium and difficulty in brainstorming all the possibilities for varied and complex situations; it is often not obvious how to write an optimal strategy by hand without doing significant testing throughout the process.

We present and discuss results for our deep Q-learning and baseline approaches for three different experiments, each increasing in complexity. First, we describe the combat environment.

4.1 NetHack environment

A modified version of the base NetHack game is used to test our combat algorithms. Mechanics that might alter experiment results were removed, including starvation, weight limitations, and item identification (all items are identified on pickup). We leave these issues

for future work.

The default NetHack map system is also altered. Instead of the default, randomly-generated maps, we use an enclosed 8x16 room for the combat environment, as seen in Figure 4.1. In each combat episode, the player is placed in the middle of the room, with a selected monster placed in a random position. In some episodes, the randomized player-monster distance will allow the player to have time to get ready before encountering the monster (e.g., to equip weapons and/or armor), and also give the player a few turns to perform ranged attacks before the monster enters melee range, whereas in others, the player must make quicker decisions about equipment.

An episode terminates on player or monster death, or after 200 actions (whichever occurs first). This large maximum number of actions is excessive, but was shown to perform better than a lower number in experiments.

Figure 4.1 The room used for the learning environment. The player ('@' character) starts in the middle of the room, and the monster ('a' character in this case) starts in a random position. The bottom two lines describe the player's current attributes and statistics.

NetHack also asks the player to choose a particular role or character class at the start of a game. Role determines the initial attributes of the player (e.g., strength or intelligence),

as well as some special initial properties. In our experiments, we fix the role to 'barbarian' (out of the 13 available), chosen for its ability to handle both melee and ranged weapons reasonably well. Casting spells and reading spellbooks are omitted in our experiments so the barbarian's low intelligence is not a contraindication.

4.2 Learning algorithms

We use deep Q-learning for our learning algorithm with custom rewards, states and actions, as detailed below. First, we present for contrast two trivial baseline algorithms that approximate a beginner player's actions.

4.2.1 Baseline strategies

To compare our deep Q-learning approach, we present two simple combat strategies for the NetHack environment that match up with our combat experiments, detailed later in section 4.3.

The first baseline is a precisely-tuned strategy for our test combat environment, where the player has a fixed inventory and faces against one particular monster. The baseline will use items and attack in an optimal manner, in order to compare against the performance of the DQL model.

The second baseline has access to a wide variety of items, including melee and ranged weapons, potions, scrolls, wands, rings, and some armor. At the beginning of each combat episode, it equips a random weapon from its inventory, then moves towards the monster and attacks when in range. If a ranged weapon is chosen, the agent will line up with the monster and then fire projectiles until supply is exhausted. If the monster is invisible, a random move in any direction will be attempted; moving into a monster is considered an attack, so randomly moving has a chance to attack an invisible monster.

Further, 25% of the time, instead of approaching/attacking the monster, it will attempt to use a random item from its inventory: a scroll, potion, wand or ring. This behaviour could be said to simplistically replicate that of a new player who will occasionally use a random item but spend the majority of their time on movement. There is some risk for this

baseline since some items can be harmful if used improperly.

4.2.2 Learning approach

A *dueling double deep Q-network* [WSH+16] with experience replay is used for the learning agent. A tabular algorithm may be successful in this environment, but a deep network allows for generalizability and far more compact representation. Experimentation with tabular forms and linear function approximation, such as tile coding, is left to future work. Below we discuss the main components of the MDP formulation for the learning model: rewards, states, and actions. Note that these three components contain many design choices, based on familiarity with NetHack and preliminary experimentation, but other choices are of course possible.

4.2.2.1 Rewards

Rewards, the most trivial of the three components, are given as follows: a small negative reward at each timestep to inspire shorter combat, while at episode end, a reward of 10 on monster death and -1 on player death.

4.2.2.2 States

Here we present the formulation of an abstract state space for NetHack combat. In modern reinforcement learning approaches, the state space for video games is typically generically defined over the raw pixels from the game screen, and actions defined as the basic controller inputs, such as was done in the Atari game experiments [MKS+13]. This approach is very easy to set up and can be ported to diverse environments with ease, but depends on a fairly simple notion of targeting and combat strategy.

In our approach we provide a more high-level representation of state. Doing so allows us to more easily address effects with long temporal dependencies and better handle the diversity of game elements and command options available in NetHack, i.e., to focus more directly on learning combat strategy instead of the more general context in which we would also need to learn the basics of pathing, maze navigation, and meanings of inventory and character status, all of which already have well-known and efficient algorithmic

solutions. There are also challenges associated with how information about game state is represented on the NetHack screen that would arise from using raw pixels as state. For example, different monsters can be depicted by the same symbol onscreen but range disparately in difficulty level. Furthermore, certain state information is not shown at all on the game screen – these 'intrinsic' properties are discussed further below.

Game state information is parsed from the normal game screen visible to a player. We encode as basic numerical information:

- the player's experience level, dungeon level, health, power, strength, dexterity, constitution, intelligence, wisdom, charisma and armor class, all normalized in the 0..1 range, and
- the normalized 0..1 distance between the player and monster,

along with information in (binary) vector form:

- the player's current status effects (confused, stunned, blinded, hallucinating, etc.),
- the player's current inventory with each ⟨item, enchantment, BUC-status, condition⟩ item tuple separately represented, and with normalized 0..1 values for projectile quantity and wand charges, and
- what the player is currently wielding and wearing.

Additionally, we include one-hot encoded vectors to represent:

- the current monster being fought (with a special category for invisible or hallucinatory monsters),
- the monster's class (according to its glyph),
- the player's character/role type (e.g., barbarian), and
- the player's alignment,

and finally, simple booleans for each of the following:

- if the player has lost health this game,
- if the player is suffering from lycanthropy,
- if the player is currently invisible,
- if the player is lined up with the monster,
- if the monster has turned invisible,
- if there are projectiles currently on the ground and if the player is standing on them, and
- if both the player and monster have recently moved or approached the other.

More details on each piece of state information can be found in Appendix A.

Note that the actual game map is not included in the state. This choice was made to significantly slim down the state space and decrease learning times. Since combat is one-on-one, and we already include monster-player distance and other combat-relevant information abstractly, it is unlikely that having the detailed game map would significantly improve success rate. In a multi-combat scenario, however, adding the game map to the state in conjunction with convolutional neural networks and longer training times could potentially lead to more clever movement tactics.

One category of information that this state space does not capture is that of character 'intrinsics.' An intrinsic is a character trait such as speed, stealth, or resistance to an element. Some intrinsics are gained temporarily from item use; for example, drinking a potion of speed will give the player the speed intrinsic. Others are obtained while wearing a piece of equipment. Intrinsics are not displayed on the NetHack screen and it is up to the player to remember what intrinsics they have been granted (although there are a handful of rare items that list current intrinsics). This recall issue is a challenge for the learning algorithm since, after drinking a potion of speed, for example, it will immediately forget that it has done so and thus will not recognize in state its new speed, potentially leading to artificially inflated action-values for the regular, non-speedy state. To get around this issue, we omit temporary intrinsic-granting items from our action set and leave a fix for future work.

One way to resolve this issue would be to maintain in state a binary vector of all items used or consumed during the current episode, or a vector of the status of all intrinsics the player currently has. However, as stated previously, some intrinsics wear off after a certain period, such as obtaining speed from the potion of speed. The duration of this type of intrinsic is randomized within a range that differs for each item, and so it is unclear when to update the vector element to indicate that the intrinsic is no longer active. It could be possible to hard-code these ranges for each such intrinsic-granting item, but that would require injecting decidedly more game-specific knowledge into the agent.

4.2.2.3 Actions

We also use an abstracted set of actions. In recently popularized reinforcement learning applications to games, the action set corresponds directly to controller inputs. In NetHack, however, basic keyboard commands can map to different actions depending on game context; keyboard characters used to access inventory items, for example, are determined by the game based on the order in which the items are acquired, more or less randomizing the key command associated with any item in each playthrough. Thus, we use abstractions in order to allow the learning agent to select an appropriate item without the expense and complexity of incorporating an entire game history into the learning process.

Our game controls are divided into basic primitives and actions mapped to inventory items. For the item actions, the action to perform depends on the item type. If an equippable item (i.e., weapon or armor) is selected as the action, then that item will be equipped when taking that action. If the item is usable, it will be used when taking the action: scrolls are read, wands zapped at the monster, and potions either quaffed or thrown at the monster (each represented separately). Further, each item is represented several times in the action set, once for each combination of its possible properties (enchantment, condition, and BUC status), since the decision to use an item strongly depends on these properties. Enchantments are capped in the [-1, +1] range to reduce complexity.

Item actions are complemented by nine primitive actions related to basic movement or other non-item strategies as follows. While the standard movement keys (‘WASD’) never change, abstracted forms are still required here since the game map is not passed in.

- Move one unit towards the monster.
- Move one unit to line up with the monster (to allow for projectiles/wands/potions to be thrown), breaking ties in position by choosing the position closer to the monster.
- Move one unit to line up with the monster, breaking ties in position by choosing the position farther from the monster.
- Attack the monster with the currently-equipped melee weapon.
- Unequip the current weapon (switch to bare hands).
- Move one unit towards the closest projectile on the ground.
- Pick up a projectile on the ground under the player.
- Move one unit in a random direction.
- Wait a turn.

4.2.2.4 Dealing with impossible actions

Many of the actions above notably have prerequisites and, at any one time, the vast majority of actions will be missing one of them. For example, a player can only use the item actions corresponding to the items they currently possess, rendering the vast majority of item actions impossible. The Q-learning algorithm must be made to handle this issue or the agent would be very unlikely to learn anything in a reasonable amount of time. Below we discuss the problem of impossible actions in general, followed by some proposed solutions.

The issue of invalid or impossible actions arises occasionally in reinforcement learning. In the standard Gridworld environment, where an agent moves on a two-dimensional grid, an example 'invalid' action is to try to move past the edge of the grid. Often, this action is accepted and silently ignored by the environment, producing no change in state. Then, with a discount factor below 1, or with negative rewards per timestep, the agent will learn that this action simply wastes a turn. In other cases, an agent taking an invalid action will immediately be given a large negative reward, without the action actually being passed on

to the environment [Res14]. This immediate reward will let the agent learn faster about which actions not to take. Sometimes an episode will terminate immediately on taking an invalid action.

In the majority of these cases, an invalid action can still be passed to the environment since there is still a means of executing the action. For example, a physical robot has the means to walk into a wall, just like a human player can try to walk into a wall. However, when dealing with more abstract actions such as 'move towards monster', certain preconditions must be satisfied (i.e., the monster must exist) or else the action has no meaning and cannot be taken. In the NetHack context in particular, there are three types of impossible actions: those that can be taken but do not result in the intended consequence (e.g., trying to pick up a projectile when none exists, which wastes a turn), those that can be taken but are rejected by NetHack with an error message without wasting a turn (e.g., trying to unequip a weapon when none is equipped), and those, mentioned earlier, that have no associated key to press (e.g., trying to use an item one does not possess), making them functionally impossible. In this latter circumstance the action cannot be passed to the environment and must be caught and dealt with beforehand. Even the middle case is disastrous, since it breaks the assumption that one action equals one turn, thus negatively affecting the action-values.

It might seem logical here to use the strategy of giving a negative reward for impossible actions. This strategy would make the agent learn to not take these actions. In the case of the NetHack environment however, which has a large action set with often very few possible actions for each state, a large amount of learning time would be consumed by taking invalid actions and receiving negative rewards. Further, when using an experience replay buffer (e.g., with a deep Q-network), a significant part of the buffer would be devoted simply to these experiences, necessitating significantly longer training times and buffer sizes to learn anything.

Instead, we alter the regular Q-learning algorithm to avoid the problem of invalid actions. To begin, we make three assumptions. First, we assume that the possibility of an action in a certain state (whether it can be taken or not in that state) can be derived solely from the state itself. For example, in order to take the action 'equip sword' in a state s, s must include information as to whether the agent currently possesses a sword, as well as any other preconditions necessary to equip the sword. Second, we assume that a prereq-

uisite function $v\_a(s)$ exists for every action and returns a boolean indicating whether the action can be taken in that state or not. With this set of functions, we can at all times narrow the action set to that of the possible actions. Thirdly, we assume that it is never possible to end up in a state with no possible actions.

With these assumptions, we then make the following two changes. First, in whatever behaviour policy used (e.g., ε-greedy), only possible actions can be selected. In the case of ε-greedy, if a random action is to be selected, invalid actions are given a probability of 0; if a greedy action is to be selected, the arg-max is taken over action-values corresponding only to possible actions. Second, a change is made to the action-value update: the max over the action-values for the next state ($\max_a Q(s_{t+1}, a)$) is taken over the set of possible actions instead of the complete action set. These two modifications eliminate the problem of impossible actions. If using experience replay, the (state, action, reward, next state) tuple must also be augmented by the list of possible actions, so that the action-values of batch updates can be similarly affected.

The necessity for the second change, which affects the action-value update, may not be immediately obvious. If all action-values are by default initialized to 0, then in the tabular case, taking the max over the whole action set would likely not result in any deleterious effect, since (depending on the reward structure of the environment) the action-values for impossible actions, which are always 0 (since the actions are never taken due to the policy restriction) will likely never exceed the action-values of possible actions. In the function approximation case, however, although an invalid action may never be taken, it could be grouped together with other states and so have an incorrectly inflated value which could end up dominating the other action-values in the max of the update equation.

The formulation of an action validity function is similar to the 'initiation set' of the options framework [SPS99]. An option is a structure to enable temporally-extended actions; each option has a policy (what to do when following the option); a termination condition (in which state the option should be stopped); and an initiation set I. An option can only be taken in a state s if $s \in I$. Since options are a generalization of primitive actions, we can think of the action 'equip sword' as an option where the initiation set for the option o corresponds to the output of the action validity function v_o.

The concept of limiting the action space based on information contained in state has

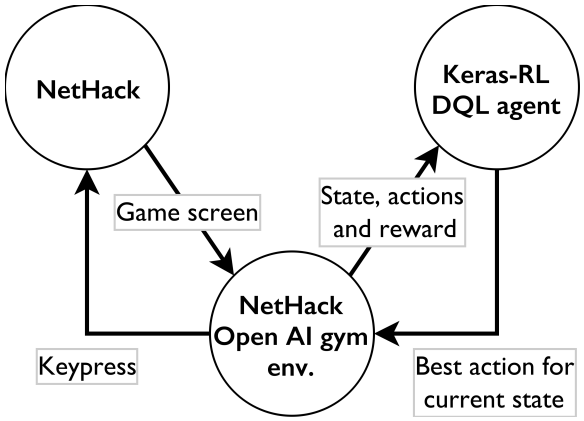

Figure 4.2 Framework for the combat learning experiments, showing interplay between the NetHack base game, the OpenAI gym environment that parses game screen information for the hand-crafted state and action set, and the Keras-RL deep Q-learning (DQL) agent.

also been referred to as a 'state-conditional action space.' Resende brings up this concept and discusses its disadvantages, namely that such a validity function must be specified, thus requiring more implementation and domain knowledge than other approaches [Res14].

4.3 Experiments conducted

Three general experiments were conducted with the player in arena-style, one-on-one combat against a selection of monsters. The first two experiments find the player battling one monster; one with a very limited selection of items as a verification of the general approach, and the second with a larger selection to demonstrate the ability of the model to choose items well. The third experiment has the same large selection as the latter but is trained on a larger number of monsters. Below we discuss the particularities of the experiments, including monster and item choice. We finish the section by detailing the model hyperparameters.

Each model will first be trained on the selected monsters, then tested on the same range. In each episode, the player is given a randomized inventory (except for the first test experiment), so no two episodes are likely to be the same. Further, randomness exists in attack chance, damage, initial monster position, and some item effects outcomes.

All experiments use the *Keras-RL* deep reinforcement learning library for deep Q-learning algorithm implementation, which is built on top of the underlying neural network framework Keras [C+15]. The Keras-RL DQL agent communicates with our learning environment, built under the *OpenAI Gym* framework [BCP+16]. This environment communicates via sockets with NetHack as described in section 2.1.3, receiving the raw game screen and sending back the command to use. It parses information from the game screen and creates the state and set of possible actions, then passes them to the DQL agent. The flowchart for the environment is shown in Figure 4.2.

4.3.1 Fire ant test

The first experiment is designed to verify the learning approach on a simple use case. Here, the player will face against one monster, the fire ant, and will be given certain items to help defend against it.

The fire ant is a fearsome sight to an unprepared player. It has two damage-dealing attacks that it deploys simultaneously to a player within range: a physical attack as well as a fire attack. The fire attack, in addition to causing damage, also has a chance to set items in the player's inventory on fire, rendering less effective several types of armor and disintegrating scrolls. Further, in this experiment dungeon level is set to 10, which slightly increases the fire ant's toughness vis-à-vis the player.

The player is given the following inventory to combat the fire ant: a dagger, tsurugi, wand of cancellation, wand of locking, and wand of make invisible. The optimal strategy would be to equip the tsurugi (which deals more damage than the dagger), line up with the fire ant, zap the wand of cancellation at it (which disables its fire attack), and then proceed to approach and attack the fire ant with the tsurugi. The other items are useless – particularly the wand of make invisible, which if zapped at the monster, will surely lead to a bad end.

The goal of this simple experiment is to verify that the DQL approach can find the correct pathway to defeat the fire ant. The items are always fixed here with no randomness. A baseline that follows the optimal strategy was also made to compare against the model.

4.3.2 Lich test

In the second experiment, the player faces against the lich, a difficult monster that can cast spells, including cursing items in the player's inventory, rendering them ineffective, or turning invisible, protecting them from some attacks. Here the player has access to a much wider range of items than earlier. The set of items sampled for inventory is discussed more in the next section, and the full list can be seen in Appendix B.

The goal of this experiment is twofold: to show that the DQL model will use items correctly against a difficult monster, and to function as a sanity check for the next experiment, which uses the same item sampling but is trained against not just one but several monsters. That set of monsters includes the lich, so we would expect to see approximately the same success rate against the lich for both this and the next experiment; a lower success rate in the wider monster experiment may indicate inadequate training time or other complications.

4.3.3 Representative model

The final experiment sees the player in combat against a monster chosen from a more diverse selection – a slice of monsters from levels 14 to 17, with some exclusions – and equipped with a larger set of items.

In particular, twelve monsters were selected from levels 14 to 17 for the player to face, each having unique abilities and requiring diverse strategies to defeat. Some monsters curse items in the player's inventory or cause their weapon to deteriorate while others have powerful melee attacks. A subset of monsters was chosen in place of the entire monster set in order to lower the computation time needed to learn a correct model, although it is likely that the agent would perform on average equally well on the entire set if trained on them, give or take some additional state information that may be required for these cases.

The exclusions from the 14-17 level range are made up of unique monsters, non-randomly-generated monsters, and shape-changing monsters. Unique or non-randomly-generated monsters (10 in the range) appear only occasionally or in certain circumstances (e.g., shopkeepers) and typically require very specialized strategies, making them a lower priority for which to learn combat. Shape-changers (2 in the range) are excluded since

monster type is expected to stay constant in an episode.

Dungeon level for these experiments is set to 1. Dungeon level does not influence monster level here, since we disable the influence it usually has over the latter.

In terms of inventory, the player is given a random sample of items to mimic the inventory of a real player facing against a monster midway through the game. The sample includes one melee weapon made of wood, one of iron, and one of silver; one ranged weapon; and 10 pieces each of two different ammunition types for use with the ranged weapon. It also includes three random potions, three random scrolls, three random wands, five random rings, and four random pieces of dragon scale mail; in total, 18 items out of a possible 102 from these categories. All items are generated as uncursed and without enchantment (+0). Items which require special input behaviour (e.g., wand of wishing which requires a string input) are excluded, as well as unique 'artifact' items which are typically over-powered. The full listing of possible items and explanations for omissions can be found in Appendix B. Note that if the agent somehow comes across an item that was not in their original inventory, or if an item in their inventory is modified (e.g., change in BUC status or enchantment), the agent will still be able to recognize and use it (as long as it is not one of the excluded items). In such case, the item will be represented by a different action (as described in section 4.2.2.3) to enable the agent to distinguish between the changed properties.

The relatively large sample of items is given to demonstrate the ability of the agent to choose suitable items for each particular monster, in addition to giving the agent a chance to win against the monster (since failure will probably result from having no items). Armor in general was omitted from the experiment since it is more useful in a general sense for combat and does not have much influence on strategies against particular monsters (the quantity of armor one has equipped is in general better than the quality). To compensate for the lack of armor, the agent is given a fixed armor class (a value related to quantity of equipped armor) of -15 at the beginning of every episode in order to simulate a reasonable amount of armor being equipped. Some armor, however, bestows additional beneficial effects upon the player that can have an impact against certain monsters. To simulate these effects, five random pieces of dragon scale mail (out of a possible 9) are given to the player in each combat. Each piece of dragon scale mail, when equipped, gives the player

a different benefit: green dragon scale mail gives resistance to poison, while the same of a silver colour makes ranged or magical attacks bounce off the player and rebound upon the hostile monster.

4.3.4 Model hyperparameters

The following hyperparameters are set before each experiment: player experience level, total number of actions for training, and neural network architecture.

Player experience level determines the starting health of the player and their chance to hit in combat (as well as other effects with no combat impact for our experiments). In our experiments, player level is automatically set based on monster difficulty in order to keep combat difficulty relatively constant. In the first experiment, the player level is set to three less than the fire ant difficulty, to emphasize the importance of using the wand of cancellation (which halves the damage output of the monster). For the two other experiments, it is also set to three lower than the monster difficulty, since the fixed armor class given increases survivability in general. In the end, these settings are somewhat arbitrary; other parameterizations are of course possible.

The total number of actions for learning should correspond to the size of the state space to be explored. For the fire ant case, the number is set to 40,000 actions or approximately 2,000-4,000 games, if each game lasts between 10-20 actions. The lich test is run over 200,000 actions. The full items case, which enables a much larger portion of the state space to be visited, is trained over 2,000,000 actions.

All models use an epsilon-greedy behaviour policy with epsilon linearly annealed from 1 to 0 through the course of training (with a value of 0 used for evaluation). Experience replay buffer size is 40,000 for the fire ant test, 200,000 for the lich test and 1,000,000 for the third case. Discount factor is 0.9.

The architecture of the neural network is as follows: an input layer of size equal to state length, followed by two dense layers, and an output layer of size equal to the number of actions. In the fire ant case, the two dense layers have 32 and 16 units respectively, while the lich test has layers of size 32/32, and the full items model size 64/32. Again, these are somewhat arbitrary values, but are determined keeping in mind that more complex

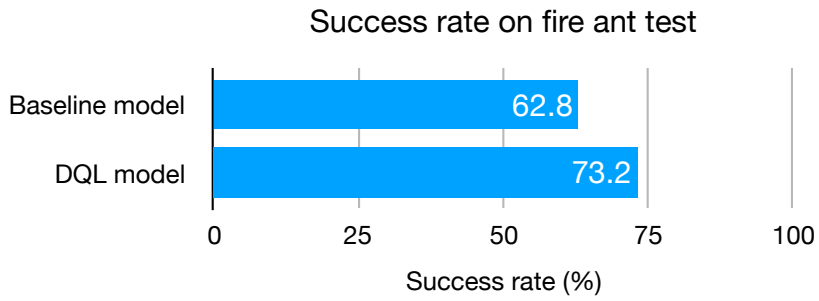

Figure 4.3 Percentage of episodes in which the baseline and DQL models succeeded against the fire ant, during testing over 500 episodes.

environments may require larger network sizes. Adam was used for the optimizer with a learning rate of 10^{-6}.

4.4 Results

We present results of the three models below: the two verification tests against the fire ant and lich as well as the model encompassing more monsters.

4.4.1 Fire ant test

Results for the baseline and DQL models on the fire ant experiment are presented in Figure 4.3. Somewhat surprisingly, the DQL model outperforms the baseline by about 10%. This result might be explained by the behaviour of the models when the monster spawns very close to the player at episode start (i.e., within melee range). The baseline will equip the tsurugi and zap the wand regardless of initial monster position, but the DQL model may opt not to since getting an extra attack or two may be preferable to equipping/zapping.

To further verify the behaviour of the DQL model, we match to the trajectory of each episode certain regular expressions corresponding to the optimal strategy for defeating the fire ant. The number of occurrences of each expression are presented in Figure 4.4. In the figure, we see that each episode ended with the 'attack monster' action, while no episode contained the useless 'random move' action. Regarding weapon choice, we see that the tsurugi, the more powerful weapon, was equipped in all 200 episodes, whereas the dagger

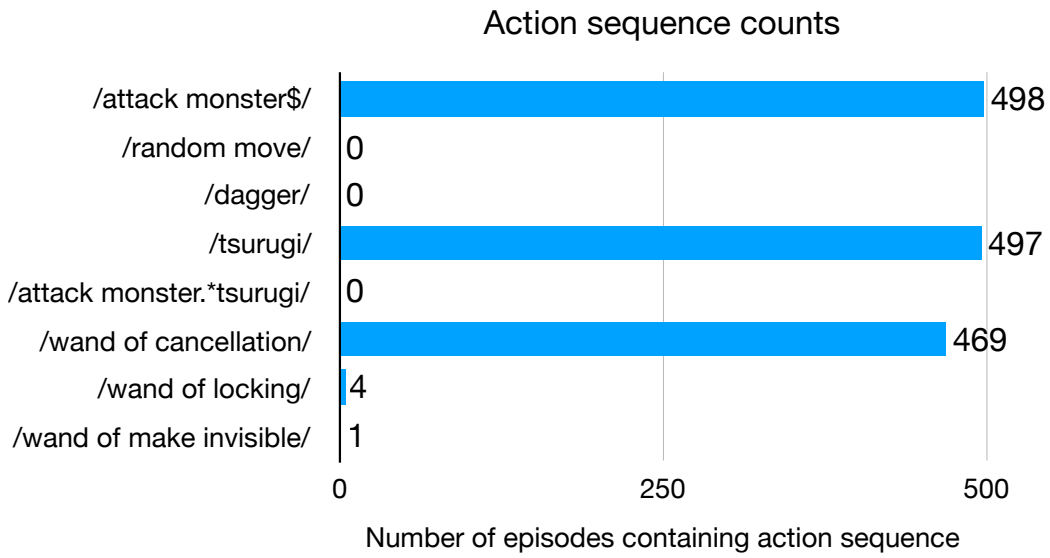

Figure 4.4 Number of episodes that contain certain sequences of actions, during testing over 500 episodes of the fire ant DQL verification model. The $ character means the action took place at the end of an episode while '.*' is a wildcard that matches any action(s) in between two others.

was equipped in none. Further, the tsurugi was always chosen to be equipped before the player began attacking the monster. We also see that in the vast majority of episodes, the wand of cancellation was used against the fire ant, whereas the useless wand of locking and harmful wand of make invisible were used a negligible amount. These results confirm that the DQL model can correctly learn a simple combat situation. It also demonstrates the difficulty to write an optimal baseline strategy that covers all situations, even in a simplistic situation.

4.4.2 Lich test

Figure 4.5 shows results for the baseline and DQL models on the lich experiment. Once again, the DQL outperforms the baseline, in this case slightly more than the fire ant test (17.25% difference versus 10.4%). Here the DQL model has the advantage of learning the success rates induced by each item, whereas the baseline always chooses a random one to use without considering possible outcomes.

The behaviour of the DQL model is further demonstrated in Figure 4.6, which lists the

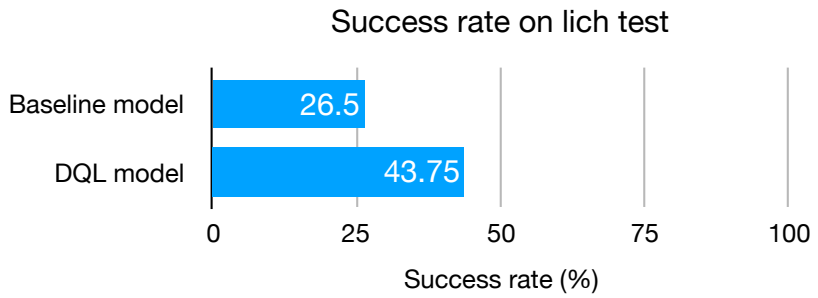

Figure 4.5 Percentage of episodes in which the baseline and DQL models succeeded against the lich, during testing over 400 episodes.

top 20 actions taken by the agent during testing, with each action counted once per episode. There is a certain bias in the figure due to the item sampling – items can only be used in the episodes in which they were sampled, whereas movement actions are available at all times and so are naturally taken in more episodes than any other. However, certain trends can be discerned. For example, the 'wait' and 'random move' actions are present in a large number of episodes, due to the lich having the ability to turn invisible and thus making regular movement actions impossible.

Another seemingly strange trend is the equipping of dragon scale mail of any kind in a large number of episodes, even though the majority of those episodes end in loss or exceed the time limit. This aberration is probably due to the fact that none of the dragon scale mail bonuses help the player against the lich, and so equipping any of them is equivalent to the 'wait' action, as described above; since five of them are generated in each episode, they would naturally occur then as some of the most taken actions. A similar case can be made for the silver spear, as well as the scrolls of light and food detection which do not lead to success against the lich. Of course, in the latter cases, there is a negative effect in using the scroll in that it may be useful at another time during normal gameplay, but the conservation of items during combat encounters (potentially through the use of negative rewards) is left to future work.

Finally, we can observe the items that do lead to success in combat: the wand of fire (deals ranged damage to the lich), ring of invisibility (enables the player to avoid attacks from the lich), and ring of increase accuracy (increases chance of attack during weapon combat). The appearance of these items in the top 20 actions, compared to the breadth of

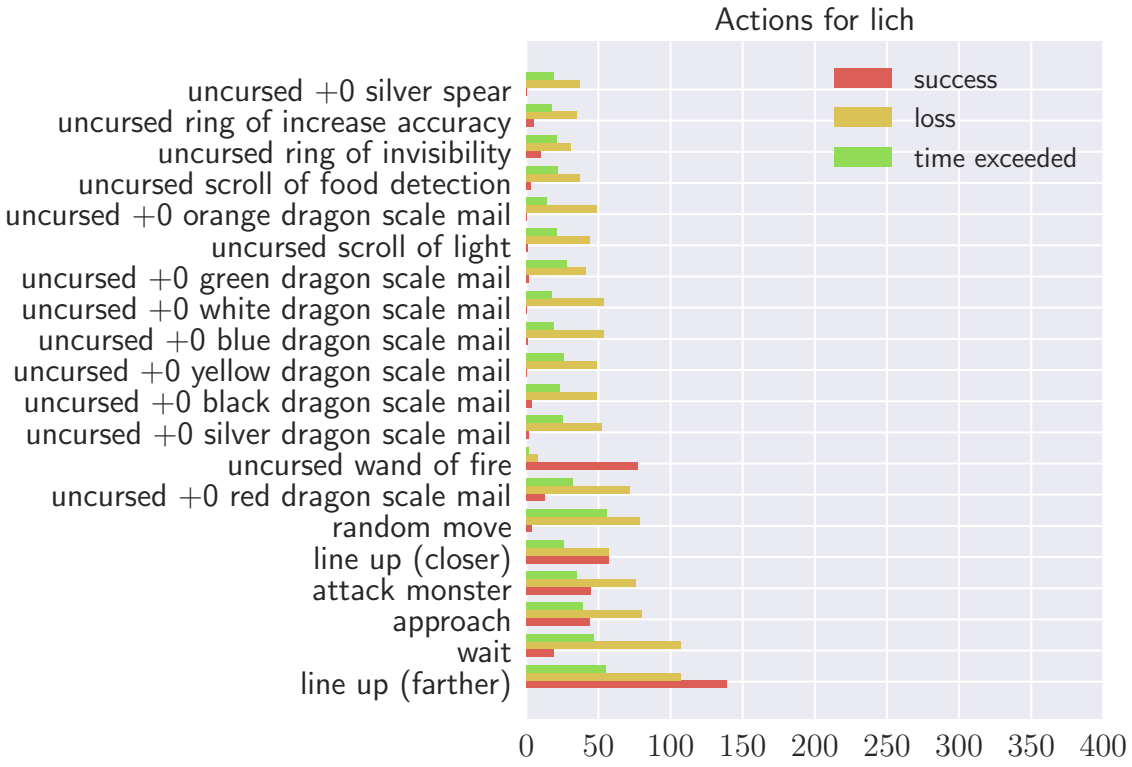

Figure 4.6 Actions taken by the DQL model against the lich in over 400 episodes sorted in order of total times taken from bottom to top. Each action is counted once per episode. Actions taken in an episode ending in success are shown as red bars and counted separately from actions taken in an episode ending in loss (yellow bars) or where the number of actions exceeded 200 in an episode (green bars).

possible items in the player's inventory, shows the aptitude of the DQL model at selecting appropriate and specialized items for this particular monster. We now turn to the model trained against a variety of monsters.

4.4.3 Representative model

Results for the model trained on the subset of monsters from levels 14 to 17 and having access to all items are presented in Figure 4.7. As seen in the figure, the DQL model performs uniformly better than the baseline. Both the DQL and baseline perform worst on the lich and nalfeshnee, both of which are difficult monsters that can curse items. Meanwhile,

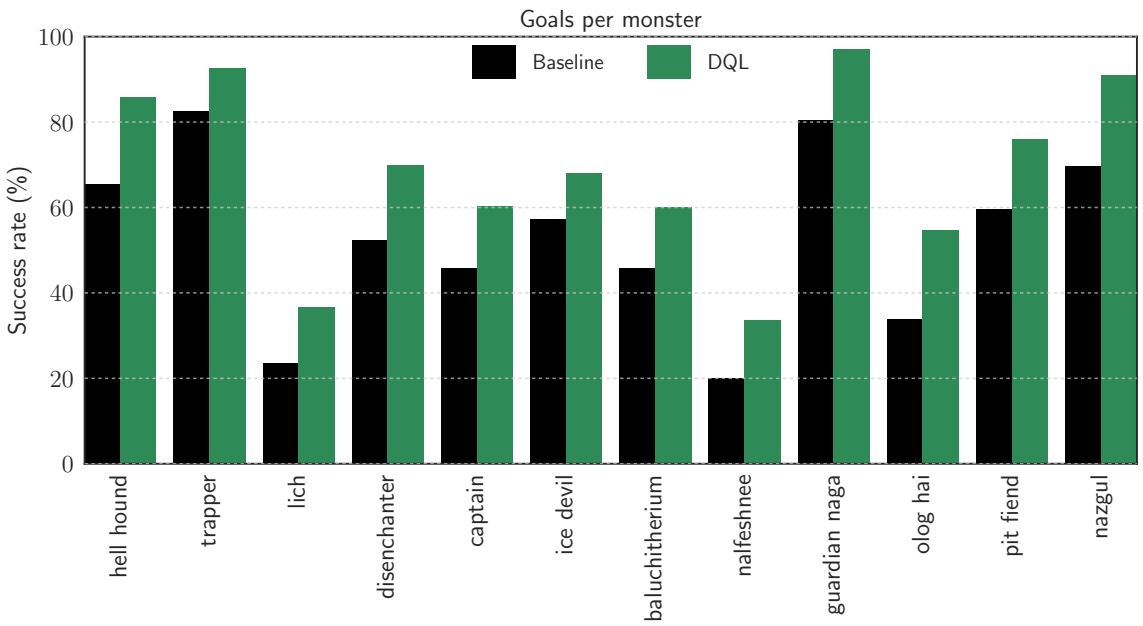

Figure 4.7 Success rates of the DQL model (lined green) vs. baseline (black) in testing on selected monsters from levels 14 to 17 with 400 episodes per monster. Vertical scale shows percentage of successful combat encounters from 0-100%.

monsters like the guardian naga or Nazgul that have powerful paralyzing attacks (whose effects would only show up after combat has ended) are easily defeated in this scenario.

To determine the correctness of this model versus the last experiment, we can compare the success percentage on the lich monster. In the latter experiment which was trained on only the lich, the DQL model had a success rate of 43.75% whereas here the number is at about 37%. This lower number could suggest that the model requires a bit more training time; results for this model did improve slightly overall when doubling training time from 1 to 2 million actions.

To better understand which items the DQL model prefers to use for each monster, the agent's actions per monster are summarized in Figure 4.8. Each action is counted once per episode, de-emphasizing repeated attacks but allowing us to verify that singular actions like equipping or using an item are performed. Further, only actions that in total make up 2% or more of all chosen actions or 2% or more of actions taken against an individual monster are listed; the others are grouped together in the 'other' category for easier readability,

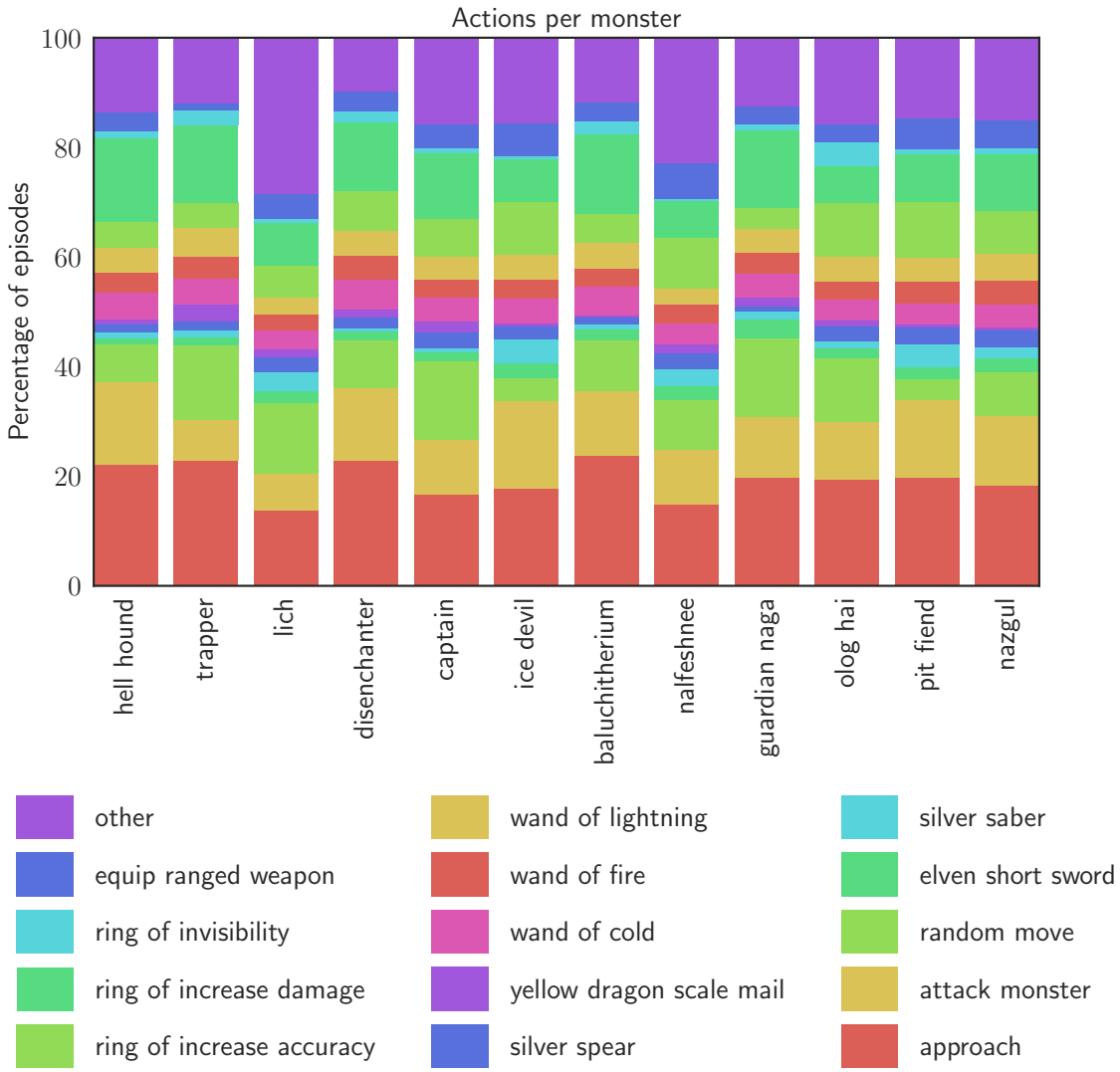

Figure 4.8 Actions taken by the DQL model in testing on selected monsters from levels 14 to 17 with 400 episodes per monster. Vertical scale shows percentage of taken actions from 0-100%. Each action is counted once per episode. The 'other' category groups together actions that are taken less than 2% of the time. Throwing any type of projectile was grouped into one category as well as equipping any type of ranged weapon. All the items present here are +0 and uncursed, omitted for brevity.

although this may prevent certain one-time use items (potions, scrolls) from being shown.

One clear indication from the figure is that the agent has learned the basic concept of approaching the monster and attacking when in range. Surprisingly, the 'line up' action, which allows the player to use wands or ranged weapons against the monster, was not taken in at least 2% of episodes, perhaps owing to the fact that lining up is less optimal than the pure approach action, which may sometimes be equivalent depending on player/monster positions.

Further, the random move action is taken in a surprising number of episodes. This result may indicate that more training time is needed, since the action does nothing but waste a turn. However, if a monster has a special ability like turning invisible (such as the lich) or engulfing the player (such as the trapper), the random move action is the only movement action available and has a chance to damage the monster, so it can have a positive effect in that (limited) circumstance.

In terms of item use, the figure shows that the DQL model prefers damage-dealing wands like that of lightning, fire, or cold to use against all monsters. In terms of rings, the most-equipped one is that of increase damage, followed by that of increase accuracy, both being rings that increase the melee and/or ranged combat effectiveness of the player. No scrolls or potions were used with any regularity, suggesting that they are underpowered compared to wands and melee attacks. With regards to weapons, two silver weapons are equipped against a number of monsters, including against those which have a special weakness to silver (ice devil, pit fiend); the elven short sword is also equipped against certain monsters, and ranged weapons are equipped against all monsters at about the same percentage. The armor chosen was in the vast majority of cases the yellow dragon scale mail, a weak choice that suggests either that no armor significantly improved combat outcomes or possibly that more training time is needed.

Of the monsters here, the lich and nalfeshnee are probably the most difficult to defeat, which could explain the larger 'other' section against these monsters. It is possible that the model is trying to use a larger variety of different items (mainly weapons) against them without seeing success on any.

Regarding number of actions taken per game, the DQL model ends up producing shorter episodes at around 30 actions on average, whereas the baseline average is over 50. The

difference here could be because the DQL model more frequently uses items like damage-dealing wands which bring a quicker end to episodes.

Finally, a note on this experiment in general: because of the large number of parameters (items, monsters, armor class, character and monster levels, and so on), it is difficult to create an experiment of appropriate difficulty. If the difficulty is too high, then there would be little difference between a baseline and DQL model since there are no optimal items to use. If the difficulty is too low, then the DQL model will not have to learn very much in order to show a large gain over the baseline. The difficulty of the above experiment may be a bit too low, demonstrated by the fact that wands are used much more than potions or scrolls, even though the latter items can contribute (albeit in a lesser role) to monster combat. Future work on this combat approach should include metrics to judge experiment difficulty leading to better experimental design.

4.5 Discussion

In this chapter, we presented an approach to learning combat in NetHack using deep Q-learning with a hand-crafted state space and action set tailored to a roguelike context. We presented results of using the DQL model on different NetHack combat environments and compared these results with corresponding baseline algorithms. Due to the limited nature of the state space, action set and environments, these results are not comparable to expert level play, but provide a useful starting framework. We now turn to using this combat approach, combined with the earlier exploration algorithms, to play a full game of NetHack.

Chapter 5
Combining Approaches

In this chapter we combine the exploration and combat approaches and apply them to a (mostly) feature-complete version of NetHack, with the algorithm exploring dungeon levels, travelling level by level through the dungeon, and fighting monsters along the way. We detail below the changes needed for each of the two components in order for them to function on the full environment. We then present results to show the success rate of the combined algorithm in terms of dungeon level depth. First, we discuss the expanded environment.

5.1 NetHack environment

To test the combined algorithm, we can use something close to the full NetHack context, but must again impose some restrictions.

We retain the presence of monsters and items here, unlike in the exploration chapter. We continue to proscribe weight limits and hunger. We also render all doors unlocked and able to be opened even if certain effects (such as the player 'fumbling') could normally prevent them from being opened. The removal of weight limits, hunger and locked doors does render the game less difficult, but still leaves a large amount of challenge for a player, autonomous or otherwise.

There are also some problematic level properties which must be addressed. First, we disallow certain *dungeon branches*: a branch is a set of optionally traversable levels that

at some point splits off from the regular level sequence. These branches (including the Gnomish Mines, Sokoban, and Quest levels) have vastly different geography, taking place in caverns with no discernible room or corridor shapes. Although there is no inherent obstacle to running our algorithm on these levels, extra implementation work is needed to recognize and parse these areas, which we do not attempt here. For the same reasons we also remove the special 'oracle' and 'big-room' levels. Late-game levels also exhibit similar behaviour, but we are unlikely to reach their depth and so they are not considered problematic at present.

5.2 Combined algorithm

We present here the combination of the aforementioned exploration and combat algorithms. Each of the two components will be engaged when necessary. The agent will begin a level by exploring, then switch over to combat upon observing a monster. On monster death, the agent will continue exploring. When exploration of the level terminates, the agent will proceed to go down the staircase to the next level and continue there. Below we discuss the changes needed for each component to function in this environment.

5.2.1 Exploration changes

Three main issues arising from the full level context must be addressed to allow the exploration algorithm to function. We consider the existence of items, the need to find the exit staircase, and the negative effects caused by monster combat that may impede exploration.

Items are often found during exploration, and due to their occasional usefulness in combat, should be picked up. Items can be randomly generated in rooms, and also drop from monsters on death. Since weight restrictions have been removed, all items can be picked up without regard for their quality or importance in combat; during combat, the DQL model will choose optimally amongst the available items, so possessing useless items is not a problem.

The methodology for integrating item pickup into the exploration algorithm is as follows. When we enter a room, the positions of all items inside are added to the frontier list

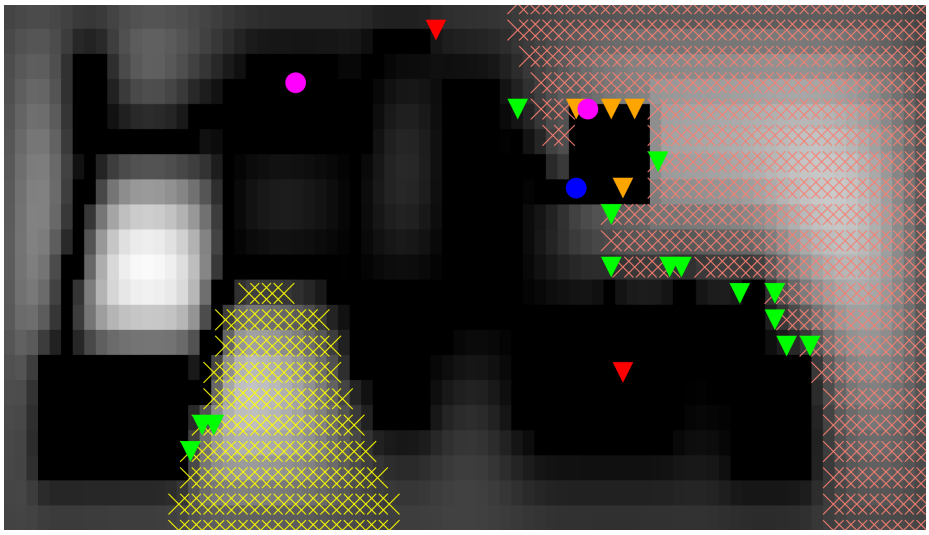

Figure 5.1 Visualization of a NetHack occupancy map with monsters and items. Monsters are represented as magenta circles and items as orange triangles. In this map, the player (blue circle) has just entered a room containing one monster and a few items.

(items in shops are ignored, due to additional complexity involving buying the item after picking it up). Then, in component evaluation, we prioritize item positions above all other frontiers, moving to and picking up each item in turn (in order of closest distance to player), before reverting to the standard component/frontier evaluation. When in a corridor, we will pick up an item if we happen to be standing on it, but to conserve actions we will not make a special diversion to other corridor spots just to pick up an item. A sample occupancy map with items highlighted is shown in Figure 5.1.

Another issue is locating the staircase to the next level, which is necessary to progress further in the game. With some occupancy map parameter settings, discovery of all rooms is not always achieved, and so the room containing the staircase may not be found. In these cases, when the exploration algorithm terminates (when there are no more interesting frontiers/components), we will continue visiting frontiers on the frontier list (in order of closest distance to player) until the room with the exit is found. If the exit has still not been found, we will start travelling to any items that have not been picked up, and then to any sleeping monsters that are still alive, since they may be obscuring the exit.

Finally, certain effects the player may contract during monster combat are deleterious to

exploration and thus must be addressed. For example, if the player is blinded, hallucinating or stunned, they will not be able to move around the level and/or accurately sense their surroundings. In these cases, we will simply wait by skipping turns until the status effect has lifted. In other cases, a monster may engulf the player, removing their view of the map; we handle this case by stopping occupancy map updates until the player is no longer engulfed. Finally, the past presence of monsters and/or items in rooms can cause difficulties in the algorithm's parsing of the map by obscuring ground tiles, leading to inaccuracies in the occupancy map. To handle these cases, we edit the NetHack code to emit the ground character (e.g., room symbol, corridor symbol, etc.) for the player's current position.

5.2.2 Combat changes

The main challenge with the combat approach is deciding when to start and end an episode. In the regular arena training case, we always receive a special signal from NetHack on monster death, so we know exactly when each episode ends. It is more ambiguous here, however, since we are progressing through a level and will have many combat encounters, and may not receive such a signal. Also, there is a much higher chance here of encountering more than one monster at once, raising additional issues to deal with if such a signal would be introduced.

A monster combat in this context will begin upon first observing a monster's glyph on the NetHack map. There are four requirements that must be met upon observation: the monster's position has to be reachable from the player's position with the player's current knowledge of the map, the monster must be within 6 units of the player (calculated using Manhattan distance), the monster must not be peaceful, tame, or a shopkeeper, and the player must be able to sense the monster using the 'detect monsters' command which outputs the names of nearby monsters. These four requirements usually guarantee that the monster can be engaged in melee or ranged combat.

Determining end of monster combat is a bit trickier. Since we are not sent a signal by NetHack when a monster dies, we must rely on the presence of the monster glyph on the map. When it disappears, we end the episode. This formulation presents some difficulties since a monster may simply walk out of the player's field of view, teleport to another place

on the map, or become invisible, yet still be considered dead. At present, we ignore these latter cases since they are each very tricky to handle.

The same state space as described in the combat section is used for these combats. We expect the monster glyph property to be especially helpful, since encounters with monsters in the early game during training could aid generalizability by helping to influence actions on monsters of the same class only encountered during testing.

5.3 Experiments conducted

We run the combined algorithm in the NetHack environment in phases, with each phase containing both a training and testing component. First, the algorithm is trained on combat encounters recorded during the previous phase, in arena-style one-on-one combat. Then, it is tested on the full NetHack context. We explain below the motivation for and details of this approach.

Training on previously-encountered monster combats is crucial to success of the combined algorithm. Since this algorithm will always start on level 1, it will be running into a small subset of monsters that get generated primarily on early levels, in addition to finding a similar small subset of early-game items, before it dies to a monster that it has never seen or strategized for. Thus, only a small portion of the state space will be visited. After training on these encounters, the agent will in testing doubtlessly get to further levels than before with its better knowledge. In these further levels, it will encounter larger and larger subsets of monsters and items. Thus it will need to be trained again on these new encounters.

To be able to train on previously-encountered monster combats, information about each combat encounter is recorded. In particular, everything necessary to reconstruct the initial state of the combat (player attributes, inventory, monster, dungeon level, etc.) is saved so that it can be reproduced exactly during training. The reconstructed state in training is then checked for equality with the recorded state to ensure correctness.

Training is performed on a sample of the recorded combats. Combats are sampled to balance out encounters between different monsters. Some monsters are much rarer than others on certain early levels, so encounters are capped at 300 per monster to address this imbalance. Duplicate encounters, based on monster name, player inventory, experience

level, strength, dexterity, armor class, status effects and dungeon level, are also removed. Further, of the 300 encounters per monster, the number in which the player is suffering a deleterious status effect (e.g., lycanthropy or confusion) is capped at 15% of total encounters. This balancing is necessary to ensure that the majority of training focuses on the most likely pathway an agent will take through the game.

We also impose a special order on the order of combats played through during training. Combats are sorted by monster difficulty: monsters of easier difficulty will be given to the agent first, followed by ones of harder difficulty. This order will help the agent learn basic combat maneuvers on easier monsters in order to save time when training on harder monsters later on that require more advanced tactics. After training on the full set of sampled combats, the same batch is then repeated twice, for a total of three times per combat. Repeating in this manner will help the agent to better explore the possible actions in each combat and refine its action-values.

5.4 Results

Results for the combined exploration/combat algorithm are presented in Figure 5.2. The figure shows how far in terms of dungeon level each successively-trained model was able to reach. As mentioned earlier, each successive model is trained on the monsters encountered during testing of the previous model; later models ('phases') thus have an advantage over earlier ones. Phase 1 is trained on the monsters encountered by the baseline algorithm, a combined baseline combat/standard occupancy map algorithm. All models were tested on 200 playthroughs of the described NetHack context. Note that due to the restrictions imposed on the environment, the only cause of player death is monster combat.

In the figure, we see that there is a general downwards trend as dungeon level increases. This trend is most probably due to two reasons. First, harder monsters are naturally found deeper in the dungeon, thus inducing more player deaths. Second, reaching a deeper dungeon level means not failing on any combats in any of the previous dungeon levels. Thus, the further one goes, the higher the probabilities of dying increase.

We also see in the figure a very large gap between the phase 1-2 models and phase 3+ models. This discrepancy is due to some altered hyperparameters. In particular, we

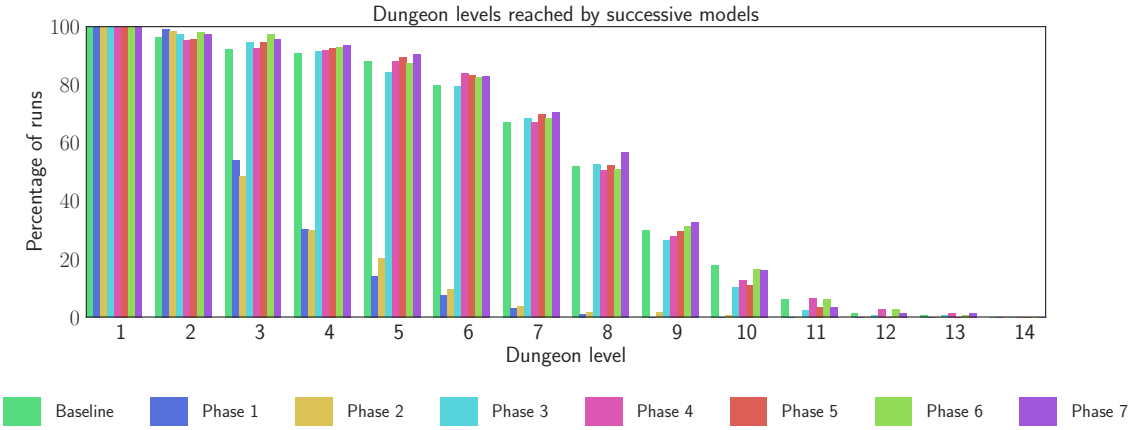

Figure 5.2 Dungeon levels reached by successively-trained models of the combined exploration/combat algorithm. Baseline uses occupancy map algorithm for exploration and baseline algorithm for combat.

decreased the learning rate from 10^{-6} to $5 * 10^{-7}$, which seemed to soften Q-network instability. We also modified the Keras-RL learning agent implementation to train over a specified number of episodes (i.e., three times the number of combat encounters) instead of a specified number of actions (which would need to be estimated based on the average actions per encounter). These two changes seem to drastically increase model success rate.

After that improvement, we see much smaller but positive gains in depth reached from phases 3 to 7, with phase 7 in particular doing better than baseline at reaching dungeon levels 7 through 9, but losing the advantage at dungeon levels 10+. It is hypothesized that further phases would continue to show gradual improvement. However, their success is limited due to their relatively simplistic combat movement actions. Future improvements to the combat algorithm may show higher gains relative to baseline.

5.5 Discussion

In this chapter we presented a combined exploration-combat algorithm for use with a nearly complete game of NetHack. Ideally, these two components would be learned concurrently with the use of perhaps the same learning algorithm, but at great computational cost. We thus demonstrated how to fuse the more computationally-friendly approaches presented

earlier in the thesis to achieve a combined algorithm. Results were presented to show the algorithm's success rates: it does moderately well but improvements can still be made. In future we would hope to make comparisons to and surpass existing rules-based bots for NetHack, which we discuss in the next chapter.

Chapter 6
Related Work

Below we will discuss related work for the exploration and combat aspects of this work, in terms of both other research as well as approaches taken by existing NetHack bots, including BotHack (the first bot to complete the game) and the Tactical Amulet Extraction Bot (TAEB). (Other NetHack bots can be found at [MLO+15]). As previously stated, these bots tend to use hard-coded strategies for NetHack mechanics.

6.1 Exploration

Automated exploration or mapping of an environment has been frequently studied in several fields, primarily including robotics and with respect to the problem of graph traversal, with the latter having some connections to video games.

Exploration in robotics branches into many different topics, with some factors being the type of environment to be explored, amount of prior knowledge about the environment, and accuracy of robotic sensors. One frequently-discussed approach is simultaneous localization and mapping (SLAM), where a robot must map a space while keeping precise its current position inside said space. Since in NetHack we always have accurate information about player position, this issue can be avoided. Good surveys of robotic exploration algorithms, with some coverage on the SLAM issue, can be found in [JGR12] and [LaV06]. A more general survey of robotic mapping that discusses exploration can be found in [Thr02]. Work involving occupancy maps in particular was presented earlier in section 2.2.

The exploration problem in robotics is also related to the coverage path planning problem, where a robot must determine a path to take that traverses the entirety of a known space. A cellular decomposition of the space is used in many such approaches. For example, Xu et al. presented an algorithm to guarantee complete coverage of a known environment containing obstacles while minimizing distance travelled, based on the boustrophedon cellular decomposition method, which decomposes a space into slices [XVR14]. See Choset [Cho01] for a comprehensive discussion and survey of selected coverage approaches.

Many exploration strategies, including our own approach, involve a selection of frontiers to visit. This selection often uses an evaluation function which takes into account objectives like minimizing distance travelled (i.e., choosing the closest frontier) or exploring the largest amount of map the fastest. Yamauchi described a strategy using occupancy maps to always move towards the closest frontier in order to explore a space [Yam97], with a focus on how to detect frontiers in imprecise occupancy maps. Gonzàlez-Baños and Latombe discussed taking into account both distance to a frontier and the 'utility' of that frontier (a measure of the unexplored area potentially visible when at that position) [GBnL02], also taking into account robotic sensor issues. We use a similar cost-utility strategy for our evaluation function, with utility determined by probabilities in the occupancy map, and cost by distance to player. Juliá showed that a cost-utility method for frontier evaluation explores more of the map faster than the closest frontier approach, but in the end takes longer to explore the entire map than the latter since it must backtrack to explore areas of low utility [JGR12]. Further discussion and comparison of evaluation functions can be found in [Ami08].

There has also been research on the idea of moving towards areas of high utility while avoiding frontiers in low utility areas. A strategy by Newman et al. used environment features (doors, wall segments, etc.) to guide autonomous exploration, with specific preferences for moving to large open areas while avoiding past areas [NBL03]. Another approach by Hernández & Baier focused on 'depression avoidance,' a property that guides real-time search algorithms to cleanly exit regions with 'heuristic depressions,' regions of the map where the heuristic used in search gives improper values when compared with heuristic values for the region's border [HB11].

Exploration can also be formulated as a graph traversal problem. An obvious correspondence exists with the travelling salesman problem. Kalyanasundarum and Pruhs described the 'online TSP' problem as exploring an unknown weighted graph, visiting all vertices while minimizing total cost, and presented an algorithm to do so efficiently [KP94]. Koenig analyzed a greedy approach to explore an unknown graph (always moving to the closest frontier) and showed that the upper bound for worst-case travel distances for full map exploration is reasonably small [KTH01, TK03]. Hsu and Hwang demonstrated a provably complete graph-based algorithm for autonomous exploration of an indoor environment [HH98].

Graph traversal for exploration can also be applied to video games. Chowdhury looked at approaches for computing a tour of a fully known environment in the context of exhaustive exploration strategies for non-player characters in video games [CV16]. Baier et al. proposed an algorithm to guide an agent through both known and partially known terrain in order to catch a moving target in video games [BBHH15]. Hagelbäck and Johansson explored the use of potential fields to discover unvisited portions of a real-time strategy game map with the goal of creating a better computer AI for the game [HJ08]. Our work, in contrast, focuses on uneven exploration in sparse, dungeon-like environments, where exhaustive approaches compete with critical resource efficiency.

BotHack, a successful autonomous player for NetHack, explores a level by preferring to move towards large unexplored spaces. It uses the Dijkstra algorithm with edge weightings that take into account how dangerous a path is, measured by the presence of monsters and/or traps along the route, items in the player's inventory that may allow for safer travel, and if the player had travelled that route without harm previously [Kra15a].

6.1.1 Secret areas in video games

Secret areas have not been widely studied. They have sometimes been mentioned in relation to their purpose in game design. They can be a mechanism to reward players for thoroughly exploring an area, occasionally containing valuable rewards that can help the player on their journey [HW10]. In certain genres, secret areas are irrelevant to player power but confer a sense of achievement on the player clever enough to find them. Gaydos & Squire found

that hidden areas in the context of educational games are memorable moments for players and generate discussion amongst them [GS12]. In contrast, procedurally-generated secret areas like the ones found in roguelike games (like NetHack) have no educational benefit and seem to invoke less excitement since the searching process becomes repetitive.

BotHack employs a simple secret area detection strategy to find secret areas in NetHack. If either the exit to the next level has not yet been found and/or there is a large rectangular chunk of the level that is unexplored and has no neighbouring frontiers, it will start searching at positions adjacent to that area [Kra15a, Kra15b]. Meanwhile, TAEB searches near unexplored areas of size at least 5x5 as well as dead-end corridors, with a maximum of 10 tries at each searchable position [MLO+13].

6.2 Learning

There has been much recent interest in applying reinforcement learning to video games. Mnih et al. notably proposed the Deep Q-Network (DQN) and showed better than human-level performance on a large set of Atari games such as Breakout and Pong [MKS+13, MKS+15]. Their model uses raw pixels (frames) from the game screen for the state space, or in a formulation proposed by [SM17], game RAM. We use the same general deep Q-learning approach here, but with a hand-crafted game state and action set to speed up learning, given the much larger basic action set and complex state space typical of roguelikes. Kempka et al. demonstrated the use of deep Q-learning on the 3D video game Doom, similarly using visual input [KWR+16], with applications by [LC17]. Narasimhan et al. used a deep learning approach to capture game state semantics in text-based Multi-User Dungeon (MUD) games using a LSTM-DQN [NKB15]. Uriarte & Ontañón presented a method for learning combat tactics for real-time strategy games from previously-collected player combat replays [UO15]. They used StarCraft for their testing environment and learned from the replays a forward model for Monte-Carlo Tree Search.

Our work uses an abstracted state space to ease learning time. Abstracted game states have been previously studied for RTS games [UO14] as well as board games like Dots-and-Boxes [ZLPZ15], in order to similarly improve efficiency on otherwise intractable state spaces.

RL approaches have also been applied to other, more traditional types of games. The use of separate value and policy networks combined with Monte-Carlo simulation has been shown to perform spectacularly on the game of Go [SHM+16], while using RL to learn Nash equilibriums in games like poker has also been studied [HS16].

BotHack uses movement-based strategies in combat, such as luring monsters into narrow corridors or retreating to the exit staircase to have an escape option [Kra15a, Kra15b]. It also prefers to use ranged weapons against monsters with special close-range attacks and abilities. TAEB's strategy is similar and includes the tactic of *kiting* monsters in groups, which is to draw one monster at a time away from the group with projectile attacks in order to not be overwhelmed. It also employs an item rating system to determine the best equipment to use [MLO+13].

An automated player also exists for *Rogue*, a 1980 predecessor to NetHack, called Rog-O-Matic. It uses an expert-system approach, although simple learning is enabled through a persistent store [MJAH84].

Chapter 7
Conclusions and Future Work

The allure of certain games lies in their challenging nature. This challenge often takes the form of puzzles that require logic and good memory to solve, and such qualities are by nature easy to code algorithmically. Games of the roguelike genre in particular share a common set of these challenges in their game systems, such as exploration of a game map or monster combat. In this thesis, we introduced algorithmic approaches to tackle these two listed aspects of roguelike games, demonstrating their effectiveness in the prototypical roguelike game, NetHack.

The exploration algorithm we presented has been shown to efficiently explore dungeon levels in NetHack, visiting open frontiers that are most likely to lead to undiscovered rooms while minimizing visitation of areas of low room probability, as well as taking into account the chances of secret room presence. It borrows the occupancy map data structure from the field of robotics, while using the concept of diffusion originally introduced in an algorithm for searching for a moving target. Results presented show improvement over a typical greedy exploration algorithm, particularly with regard to discovery of secret rooms.

An exploration algorithm like the one set forth can have many uses. It can alleviate the tedium of the repetitive movement needed for manual exploration if used as an occasional substitute. It can further be helpful to those operating with reduced interfaces. Exploration is also a significant sub-problem in developing more fully automated, learning AI, and techniques which can algorithmically solve exploration can be useful in allowing future automation to focus on applying AI to higher level strategy rather than basic movement

concerns.

Future work on the presented exploration algorithm aims at further increasing exploration efficiency, as well as verifying its generalization on other roguelike games and games with similar map designs. One particular avenue to increase exploration would be to further analyze the average map exploration trajectory with an eye to eliminating wasted moves towards the end of a trajectory (e.g., by better determining when to stop exploring). We would also like to further explore the concept of diffusion, perhaps considering a 'local' diffusion of probabilities that would radiate from the player's current position, in order to accommodate larger map sizes. A sharper departure would be to replace diffusion with static prior information about the typical nature of maps, thus lending more stability to the algorithm. Finally, with regards to efficiency of computation, different data structures needing to store less information may be considered, as commonly done in practical SLAM (simultaneous localization and mapping) algorithms in robotics [TBF05].

Meanwhile, our approach to combat uses a deep reinforcement learning algorithm in order to develop a model successful at matching simplistic movement tactics and a wide variety of items to the multitude of monsters in NetHack. To reduce environment complexity, the approach uses a hand-crafted state space and action set, with experiments conducted in a limited 'arena combat' setting. Combat is a significant and difficult mechanic in roguelike games, and our learning approach points towards obviating the need to hard-code strategies for an autonomous player. Results for our model show that it outperforms a simpler baseline strategy, trading long but automated training costs for better performance.

The combat approach presented deals with the basic movement and item concerns, but there are many avenues for future work. In terms of increasing success rate, more advanced movement tactics could be considered, such as retreating or escaping from monsters that are too powerful for the agent or that do not have to be killed for progression (like those that are tame or peaceful, or not immediately hostile). In addition, we would like to better maintain item resources by disincentivizing the agent from using powerful items on weaker monsters, perhaps deriving values for item utility and monster power from learned action-values. We would also like to verify the generality of the combat approach on other roguelike games, as well as investigating more complicated, multi-combat scenarios, where resource management and prolonged status effects come into play. Finally, we would like

to compare our model results against a more advanced baseline, either by running an existing rules-based NetHack bot in our combat environment, or by parsing player data from online sources and gathering their win rates in similar combat scenarios.

Finally, this thesis presented a simple combination of the exploration and combat approaches on a wider NetHack context, showing the effectiveness of learning combat in stages and its improvement compared to baseline. This combination can be improved in future by extending support for more game features; for example, allowing the exploration approach to consider more diverse dungeon layouts that sometimes occur in the game, or giving the combat approach information about nearby dungeon features like fountains or stair access that can enable more advanced tactics.

Bibliography

[Ami08] Francesco Amigoni. Experimental evaluation of some exploration strategies for mobile robots. In *Proceedings of the IEEE International Conference on Robotics and Automation*, May 2008, ICRA'08, pages 2818–2823.

[Bai95] Leemon C. Baird, III. Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the Twelfth International Conference on Machine Learning*, Tahoe City, California, USA, July 1995, ICML'95, pages 30–37.

[BBHH15] Jorge A. Baier, Adi Botea, Daniel Harabor, and Carlos Hernández. Fast algorithm for catching a prey quickly in known and partially known game maps. *IEEE Transactions on Computational Intelligence and AI in Games*, 7(2):193–199, June 2015.

[BCP+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, June 2016.
<https://arxiv.org/abs/1606.01540> .

[BFDW06] Frédéric Bourgault, Tomonari Furukawa, and Hugh F. Durrant-Whyte. Optimal search for a lost target in a Bayesian world.

In Shin'ichi Yuta, Hajima Asama, Erwin Prassler, Takashi Tsubouchi, and Sebastian Thrun, editors, *Field and Service Robotics: Recent Advances in Reserch and Applications*, pages 209–222. Springer, Berlin, Heidelberg, July 2006.

[C+15] François Chollet et al. Keras. `https://github.com/fchollet/keras`, 2015.

[Cho01] Howie Choset. Coverage for robotics — a survey of recent results. *Annals of Mathematics and Artificial Intelligence*, 31(1-4):113–126, May 2001.

[CV16] Muntasir Chowdhury and Clark Verbrugge. Exhaustive exploration strategies for NPCs. In *Proceedings of the 1st International Joint Conference of DiGRA and FDG: 7th Workshop on Procedural Content Generation*, August 2016, DiGRA/FDG'16.

[CV17a] Jonathan Campbell and Clark Verbrugge. Exploration in NetHack using occupancy maps. In *Proceedings of the Twelfth International Conference on Foundations of Digital Games*, August 2017, FDG'17.

[CV17b] Jonathan Campbell and Clark Verbrugge. Learning combat in NetHack. In *Proceedings of the Thirteenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, October 2017, AIIDE'17, pages 16–22.

[GBnL02] Héctor H. Gonzàlez-Baños and Jean-Claude Latombe. Navigation strategies for exploring indoor environments. *International Journal of Robotics Research*, 21(10-11):829–848, October 2002.

[Glo13] Glosser.ca. Colored neural network. `https://commons.wikimedia.org/wiki/File:Colored_neural_network.svg`, 2013.

[GS12] Matthew J. Gaydos and Kurt D. Squire. Role playing games for scientific citizenship. *Cultural Studies of Science Education*, 7(4):821–844, March 2012.

[HB11] Carlos Hernández and Jorge A. Baier. Real-time heuristic search with depression avoidance. In *22nd International Joint Conference on Artificial Intelligence*, Barcelona, Catalonia, Spain, July 2011, IJCAI'11, pages 578–583. AAAI Press.

[HGS16] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double Q-learning. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, Phoenix, Arizona, February 2016, AAAI'16, pages 2094–2100. AAAI Press.

[HH98] Jane Yung-Jen Hsu and Liang-Sheng Hwang. A graph-based exploration strategy of indoor environments by an autonomous mobile robot. In *Proceedings of the IEEE International Conference on Robotics and Automation*, May 1998, volume 2 of *ICRA'98*, pages 1262–1268.

[Hin11] Pieter Hintjens. ZeroMQ: The Guide. `http://zguide.zeromq.org`, 2011.

[HJ08] Johan Hagelbäck and Stefan J. Johansson. Dealing with fog of war in a Real Time Strategy game environment. In *IEEE Conference on Computational Intelligence and Games*, December 2008, CIG'08, pages 55–62.

[HS16] Johannes Heinrich and David Silver. Deep reinforcement learning from self-play in imperfect-information games. In *NIPS Deep Reinforcement Learning Workshop*, March 2016.

[HW10] Kenneth Hullett and Jim Whitehead. Design patterns in FPS levels. In *Proceedings of the Fifth International Conference on the Foundations of Digital Games*, June 2010, FDG'10, pages 78–85.

[Int08] International Roguelike Development Conference. Berlin interpretation. `http://www.roguebasin.com/index.php?title=Berlin_Interpretation`, 2008.

[Isl05] Damián Isla. Probabilistic target-tracking and search using occupancy maps. In *AI Game Programming Wisdom 3*. Charles River Media, 2005.

[Isl13] Damián Isla. Third Eye Crime: Building a stealth game around occupancy maps. In *Proceedings of the Ninth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, October 2013, AIIDE'13.

[JGR12] Miguel Juliá, Arturo Gil, and Oscar Reinoso. A comparison of path planning strategies for autonomous exploration and mapping of unknown environments. *Autonomous Robots*, 33(4):427–444, November 2012.

[KP94] Bala Kalyanasundaram and Kirk R. Pruhs. Constructing competitive tours from local information. *Theoretical Computer Science*, 130(1):125–138, August 1994.

[Kra15a] Jan Krajicek. BotHack - a NetHack bot framework. `https://github.com/krajj7/BotHack`, 2015.

[Kra15b] Jan Krajicek. Framework for the implementation of bots for the game NetHack. Master's thesis, Charles University in Prague, 2015.

[KTH01] Sven Koenig, Craig Tovey, and William Halliburton. Greedy mapping of terrain. In *Proceedings of the IEEE International Conference on Robotics and Automation*, February 2001, volume 4 of *ICRA'01*, pages 3594–3599.

[KWR+16] Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. ViZDoom: A Doom-based AI research platform for visual reinforcement learning. In *IEEE Conference on Computational Intelligence and Games*, September 2016, CIG'16, pages 1–8.

[LaV06] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006.

[LC17] Guillaume Lample and Devendra Singh Chaplot. Playing FPS games with deep reinforcement learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, February 2017, AAAI'17, pages 2140–2146. AAAI Press.

[LLKS86] E. L. Lawler, Jan Karel Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, New York, 1986.

[ME85] Hans Moravec and Alberto E. Elfes. High resolution maps from wide angle sonar. In *Proceedings of the IEEE International Conference on Robotics and Automation*, March 1985, ICRA'85, pages 116–121.

[MJAH84] Michael K. Mauldin, Guy Jacobson, Andrew Appel, and Leonard Hamey. Rog-O-Matic: A belligerent expert system. In *Proceedings of the Fifth Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, 1984, volume 5.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*, December 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[MLO+09] Shawn M. Moore, Jesse Luehrs, Stefan O'Rear, Sebastian P., Sean Kelly, Anthony Boyd, and Alex Smith. Synchronizing with NetHack. `https://taeb.github.io/2009/06/12/synchronizing-with-nethack.html`, 2009.

[MLO+13] Shawn M. Moore, Jesse Luehrs, Stefan O'Rear, Sebastian P., Sean Kelly, Anthony Boyd, and Alex Smith. The Tactical Amulet Extraction Bot – behavioral AI. `https://github.com/TAEB/TAEB-AI-Behavioral`, 2013.

[MLO+15] Shawn M. Moore, Jesse Luehrs, Stefan O'Rear, Sebastian P., Sean Kelly, Anthony Boyd, and Alex Smith. TAEB - other bots. `https://taeb.github.io/bots.html`, 2015.

[Mor88] Hans Moravec. Sensor fusion in certainty grids for mobile robots. *AI Magazine*, 9(2):61–74, July 1988.

[NBL03] Paul M. Newman, Michael Bosse, and John J. Leonard. Autonomous feature-based exploration. In *Proceedings of the IEEE International Conference on Robotics and Automation*, September 2003, volume 1 of *ICRA'03*, pages 1234–1240.

[NetHack Dev Team15] NetHack Dev Team. NetHack 3.6.0: Download the source. `http://www.nethack.org/v360/download-src.html`, 1987–2015.

[NetHack Wiki15] NetHack Wiki. Starvation. `https://nethackwiki.com/wiki/Starvation`, 2015.

[NetHack Wiki16a] NetHack Wiki. Comestible. https://nethackwiki.com/wiki/Comestible#Food_strategy, 2016.

[NetHack Wiki16b] NetHack Wiki. Standard strategy. https://nethackwiki.com/wiki/Standard_strategy, 2016.

[NetHack Wiki17a] NetHack Wiki. Black pudding — NetHack wiki. https://nethackwiki.com/wiki/Black_pudding, 2017.

[NetHack Wiki17b] NetHack Wiki. Strength in game formulas. https://nethackwiki.com/wiki/Attribute#Strength_in_game_formulas, 2017.

[NKB15] Karthik Narasimhan, Tejas D. Kulkarni, and Regina Barzilay. Language understanding for text-based games using deep reinforcement learning. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, September 2015, EMNLP'15, pages 1–11.

[pNs16] N.A.O. public NetHack server. NetHack – top types of death. https://alt.org/nethack/topdeaths.html, 2016.

[Res14] Pedro Tomás Mendes Resende. Reinforcement learning of task plans for real robot systems. Master's thesis, Instituto Superior Técnico, 2014.

[SB98] Richard S. Sutton and Andrew G. Barto. *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition, 1998.

[SHM+16] David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Ko-

ray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484–503, January 2016.

[SM17] Jakub Sygnowski and Henryk Michalewski. Learning from the memory of Atari 2600. In Tristan Cazenave, Mark H.M. Winands, Stefan Edelkamp, Stephan Schiffel, Michael Thielscher, and Julian Togelius, editors, *Computer Games: 5th Workshop on Computer Games, and 5th Workshop on General Intelligence in Game-Playing Agents, Held in Conjunction with the 25th International Conference on Artificial Intelligence (Revised Selected Papers)*, New York, NY, USA, July 2017, CGW'16/GIGA'16/IJCAI'16, pages 71–85.

[SPS99] Richard S. Sutton, Doina Precup, and Satinder Singh. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112(1-2):181–211, August 1999.

[Sut17] Kent Sutherland. Playing roguelikes when you can't see — Rock, Paper, Shotgun. `https://www.rockpapershotgun.com/2017/04/05/playing-roguelikes-when-you-cant-see`, 2017.

[TBF05] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics*. The MIT Press, 2005.

[Thr02] Sebastian Thrun. Robotic mapping: A survey. In Gerhard Lakemeyer and Bernhard Nebel, editors, *Exploring Artificial Intelligence in the New Millennium*, pages 1–35. Morgan Kaufmann, 2002.

[TK03] Craig Tovey and Sven Koenig. Improved analysis of greedy mapping. In *Proceedings of the International Conference on Intelli-*

gent Robots and Systems, October 2003, volume 4 of *IROS'03*, pages 3251–3257.

[UO14] Alberto Uriarte and Santiago Ontañón. Game-tree search over high-level game states in RTS games. In *Proceedings of the Tenth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, October 2014, AIIDE'14. AAAI Press.

[UO15] Alberto Uriarte and Santiago Ontañón. Automatic learning of combat models for RTS games. In *Proceedings of the Eleventh Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, November 2015, AIIDE'15. AAAI Press.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. In *Machine Learning*, May 1992, pages 279–292.

[WSH+16] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. Dueling network architectures for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning*, New York, NY, USA, 2016, volume 48 of *ICML'16*, pages 1995–2003.

[XVR14] Anqi Xu, Chatavut Viriyasuthee, and Ioannis Rekleitis. Efficient complete coverage of a known arbitrary environment with applications to aerial operations. *Autonomous Robots*, 36(4):365–381, April 2014.

[Yam97] Brian Yamauchi. A frontier-based approach for autonomous exploration. In *Proceedings of the IEEE International Symposium on Computational Intelligence in Robotics and Automation*, July 1997, CIRA'97, pages 146–151.

[ZLPZ15] Yimeng Zhuang, Shuqin Li, Tom Vincent Peters, and Chenguang Zhang. Improving Monte-Carlo tree search for dots-and-boxes

with a novel board representation and artificial neural networks. In *IEEE Conference on Computational Intelligence and Games*, August 2015, CIG'15, pages 314–321.

Appendix A
Combat state information

Below we describe in detail the information that makes up game state for the deep Q-learning approach to combat. Each piece of information is described with emphasis on its importance for combat, as well as size, type and any additional notes (such as implementation details).

Monster information

Description Two vectors indicating the monster currently being faced and its monster class. The elements corresponding to both monster name and class are set to 1, with others 0. If the player is hallucinating, the observed monster name/class is discarded and instead two elements corresponding to hallucinatory monster/monster class are set to 1 with others 0. If more than one monster is present (i.e., if the first monster spawns a second using a special ability), the element corresponding to the second monster name is also set to 1, while for monster class, only the class of the closest monster to the player is set to 1 (explained below). Finally, one boolean is included to indicate whether the monster has become invisible.

Importance Allows the agent to distinguish between different monsters in order for monster-specific strategies to be learned. Monster class allows for strategies to be generalized

between monsters of the same class, who often share properties (e.g., weaknesses to certain weapons).

Notes

- Monster names are retrieved by altering NetHack to automatically emit names of monsters within visible range of the player into the top line of game output. (A command also exists to emit the same information.) The names of peaceful and tame monsters as well as shopkeepers are ignored during parsing.
- When multiple monsters are present, for simplicity only the class of the closest monster is set to 1, since monster class is based on output glyph and it is easier to calculate distance between glyphs on the NetHack map than it is to get a monster position based on name. This limitation could be lifted by writing a mapping from monster name to class and then querying the class for all monsters in range.
- Monster invisibility is determined by absence of the monster glyph on the map while the episode has not terminated. In the full level context, invisibility is always set to false since combat episodes terminate on absence of the monster glyph.

Type Binary vector

Size (in units) $|monsters| + |monster\_classes| + 2 + 1$

Number of monsters

Description A categorical vector indicating how many monsters are near the player: 0 for no monsters, 1 for one monster, and 2 for two or more monsters.

Importance Allows the agent to distinguish between simple combat situations with only one monster and more complex scenarios with more than one monster.

Notes Number of monsters is calculated by counting the number of monster glyphs within 6 spaces of the player on the NetHack map. Manhattan distance is used to reduce computation time, but might introduce error in certain cases.

Type One-hot vector

Size 3

Basic character information

Description Two vectors indicating the character's role (class) and alignment.

Importance Character role has impact on combat success in some cases (the barbarian role is more skilled at using two-handed weapons, e.g.). Alignment has impact on initial neutrality of certain monsters: some will not attack you if you are of a certain alignment, allowing you to have the first attack. An extra role is added to be used if the player is lycanthropic and has transformed into a were-creature.

Notes Role and alignment are parsed from the bottom lines of the NetHack game output.

Type One-hot vectors

Size $|roles| + 1 + |alignments|$

Character attributes & statistics

Description Normalized 0..1 values for the following character attributes & statistics: character level, dungeon level, hit points (health), power (mana), armor class, strength, dexterity, constitution, intelligence, wisdom, and charisma.

Importance Each attribute/statistic can have an impact on combat. For example, strength relates to damage output, while armor class relates to defense. Some attributes (like intelligence or power) only have impact for certain classes (e.g., those that can cast spells or read spellbooks).

Notes All attributes and statistics are parsed from the bottom lines of the NetHack game output. Values are normalized to the 0..1 range using the following minimum and maximum values:

- Dungeon level: 0 - 30 (30 is an arbitrary number, but a higher level is not expected to be reached).
- Character level: 0 - 30 (30 is an arbitrary number, but a higher level is not expected to be reached).
- Hit points: 0 - maximum hit points (also parsed from the bottom lines)
- Power: 0 - maximum power (also parsed from the bottom lines)
- Armor class: -40 - 15 (limits of game values)
- Strength: 3 - 25 (limits of game values) (intermediary values from 18 to 19 are mapped to 19-21 as indicated on [NetHack Wiki17b]).
- Dexterity, constitution, intelligence, wisdom, charisma: 3 - 25 (limits of game values)

Type Vector

Size 11

Character status effects

Description A vector for the various status effects that a NetHack character can experience: stunned, confused, blinded, and hallucinating.

Importance Status effects can have a deleterious effect on combat (e.g., a confused player will sometimes move in a random direction instead of the direction input by the user). Status effects can be removed with the help of certain items (e.g., using certain scrolls or quaffing certain potions).

Notes
- Parsed from the bottom lines of the NetHack game output.
- Since hunger and weight are removed from our context, the associated status effects (starving, burdened, etc.) are not considered here.

Type Binary vector

Size $|status\_effects|$

Misc. player information

Description Three booleans that indicate other aspects of player state: whether they are invisible, lycanthropic (i.e., occasionally transform into a were-creature), or have lost health in the current combat episode.

Importance All three pieces of state have impact on combat. If the player is invisible, the monster may be less likely to land an accurate attack. If they are lycanthropic, they can at random times and without prior notice transform into a were-creature, which can significantly alter combat strategy. The player having lost health can indicate the monster is hostile rather than peaceful.

Notes Player invisibility is determined by checking if the glyph for the player is the regular '@' symbol or empty " " symbol which represents invisibility. Lycanthropy is recognized when either the message 'You feel feverish' appears in the top line of the NetHack output (which indicates the player has contracted lycanthropy) or when the player's role is of a were-creature type. Lycanthropy is remembered for the rest of a combat episode. Whether the player has lost health or not is tracked by a boolean.

Type 3 booleans

Player inventory

Description A vector corresponding to all items in the player's inventory. Items possessed are set to 1 with others 0. Wands have charges (number of times you can use them) and so are represented as a 0..1 normalized value, while projectile quantity is also normalized to the same range.

Importance The learning agent must know which items they possess in order to determine which item actions are possible and which are impossible.

Notes Inventory state is queried at the beginning of each turn (such a query does not consume a turn itself) and is sent in a special message from NetHack to the learning

agent. The names in the message are parsed to activate the correct elements in the vector.

Type Vector

Size $|items|$

Current player equipment

Description Two vectors indicating which weapon the player is currently wielding and what pieces of armor the player is currently wearing.

Importance Weapon and armor choice are paramount for success in monster combat. Certain weapons and armor are much more effective than others, based on a wide range of criteria, including monster type, player role and attributes, and more.

Notes Equipped weapon and armors are parsed from the NetHack inventory message described above (wielded/worn items have a special suffix after their name).

Type Two binary vectors

Size $|weapons| + |armor|$

Ranged information

Description Boolean values related to ranged weapon information: whether the player is in the line of fire with the monster (and is thus able to direct projectiles, potions or wands at them), whether there are projectiles lying on the floor of the current room, and whether the player is currently standing on any projectiles.

Importance All three booleans are important for ranged combat.

Type 3 booleans

Size 3

Player/monster distance information

Description Information related to distance between player and monster: the normalized 0..1 distance between them, whether the player or monster changed position in the previous turn, and whether the player or monster approached each other in the previous turn.

Importance Distance is crucial to combat strategy since a far distance lets you equip weapons/armor and throw projectiles at the monster, whereas a short distance means a melee attack may be more useful. The boolean values indicating delta movement in the previous turn can indicate whether the monster is capable of movement or not.

Notes Distance is normalized to the 0..1 range using 0 as minimum and the maximum line distance in a NetHack map as maximum.

Type Vector

Size 5

Appendix B
Combat items

In this section we list the items that can be generated in the inventory of the player during monster combat for our combat experiments.

All items are generated as being uncursed, having no enchantment (+0) and being in good condition, but the model does distinguish between blessed/uncursed/cursed items (BUC status) and -1/+0/+1 enchantments if an item in the player's inventory is changed from an external source (e.g., monster spell) since each of these BUC/enchantment/condition combinations are represented separately in the state space and action set.

Almost all weapons and ammunition types that appear in NetHack are present here. Artifact weapons (powerful, uniquely-occurring items) are excluded due to their power, as well as weapons that are not randomly generated (like the tsurugi). Polearms are also excluded due to non-trivial required implementation. In terms of usable items (potions, scrolls, and wands), several items in each category were omitted if either not very useful in monster combat (e.g., potion of object detection or scroll of mail), or if they require additional string input or other implementation when used (e.g., wand of wishing, scroll of identify, scroll of blank paper).

The full list is as follows:

Wooden weapons: elven dagger, elven short sword, elven broadsword, club, quarterstaff, elven spear.

Iron weapons: orcish dagger, dagger, athame, knife, stiletto, axe, battle-axe, pick-axe, dwarvish mattock, orcish short sword, dwarvish short sword, short sword, broadsword,

long sword, katana, two-handed sword, scimitar, aklys, mace, morning star, flail, grappling hook, war hammer, orcish spear, dwarvish spear, spear, javelin, trident, lance.

Silver weapons: silver dagger, silver saber, silver spear.

Ranged weapons: bow, elven bow, orcish bow, yumi, crossbow, sling.

Ammunition types: elven arrow, orcish arrow, silver arrow, arrow, ya, crossbow bolt.

Potions: potion of booze, potion of sickness, potion of confusion, potion of extra healing, potion of hallucination, potion of healing, potion of restore ability, potion of sleeping, potion of blindness, potion of gain energy, potion of monster detection, potion of full healing, potion of acid, potion of gain ability, potion of gain level, potion of invisibility.

Scrolls: scroll of light, scroll of confuse monster, scroll of destroy armor, scroll of fire, scroll of food detection, scroll of gold detection, scroll of scare monster, scroll of punishment, scroll of remove curse.

Wands: wand of magic missile, wand of make invisible, wand of opening, wand of slow monster, wand of speed monster, wand of striking, wand of undead turning, wand of cold, wand of fire, wand of lightning, wand of sleep, wand of cancellation, wand of polymorph, wand of death.

Rings: ring of protection, ring of protection from shape changers, ring of increase accuracy, ring of increase damage, ring of invisibility, ring of see invisible, ring of free action.

Dragon scale mail: blue dragon scale mail, black dragon scale mail, gray dragon scale mail, green dragon scale mail, orange dragon scale mail, red dragon scale mail, silver dragon scale mail, white dragon scale mail, yellow dragon scale mail.