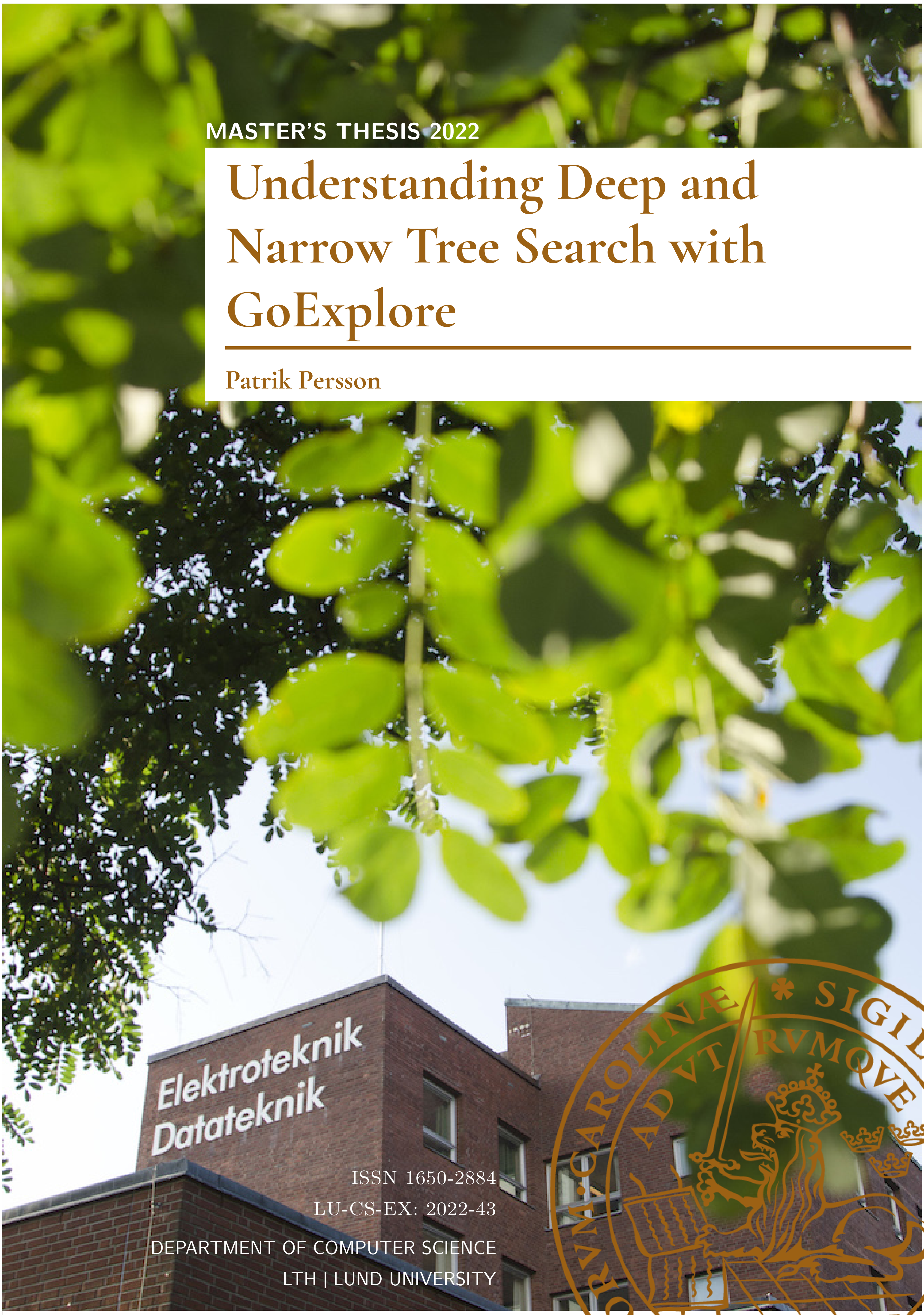
MASTER'S THESIS 2022

Understanding Deep and Narrow Tree Search with GoExplore

Patrik Persson



ISSN 1650-2884
LU-CS-EX: 2022-43
DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

Understanding Deep and Narrow Tree Search with GoExplore

Author: Patrik Persson, dat14pp1@student.lu.se
Supervisor: Volker Krueger, volker.krueger@cs.lth.se
Examiner: Elin Anna Topp, elin_anna.topp@cs.lth.se
June 29, 2022
Code available at: `https://www.github.com/papersson/goexplore`

The search space of large game trees can be reduced by restricting the depth or the breadth of the search. Most game-playing algorithms follow the first approach, where the tree is cut off at some depth and leaf nodes are evaluated according to an approximate value function that estimates the outcome of the game from this position. The approximated value function tends to be inaccurate in sparse reward environments, leading to poor overall performance. GoExplore avoids this problem by doing a deep and narrow search of the game tree, which has led to state-of-the-art results on sparse reward games such as Montezuma's Revenge and Pitfall. This thesis illustrates how GoExplore successfully drives this type of search by expanding the frontier towards novel states in the absence of rewards, and by re-focusing the frontier around rewards when they do occur. Furthermore, this thesis improves GoExplore's time complexity to linear time.

1. Introduction

Reinforcement learning algorithms have surpassed human-level performance on board games such as backgammon, chess, and Go. Recently, artificial agents have also displayed impressive results on video games such as the Atari games, Dota 2, and StarCraft 2. These algorithms can be applied to more general domains such as robotics, but games are interesting to explore for two reasons. First, due to the rules of the games, they are often quite easy to formalize mathematically. Second, they provide a natural performance benchmark since we can compare an artificial agent's performance against human performance.

One particular game that these algorithms have struggled with over the years is the Atari game Montezuma's Revenge. This is due to the game's sparse reward environment: it requires long sequences of correct actions for the agent to receive rewards, which makes learning difficult for reinforcement learning algorithms.

In 2019, the GoExplore algorithm made a breakthrough by massively outperforming the previous state-of-the-art on Montezuma's Revenge. Previous algorithms could barely outperform an average human, while GoExplore managed to surpass even expert human players. Moreover, variations of GoExplore that included domain-knowledge managed to even beat the human world record [Ecoffet et al., 2019]. Figure 1 summarizes the progress over the past decade until GoExplore's publication.

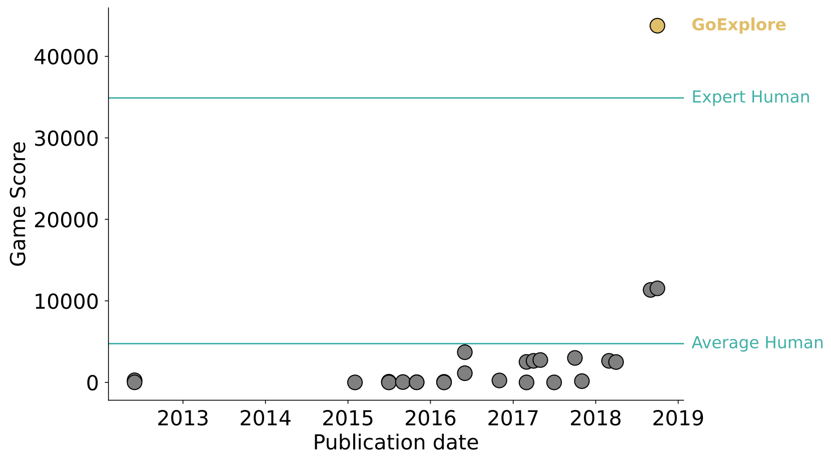


Figure 1: **Progress on Montezuma's Revenge over the last decade.** Grey markers show results from previous state-of-the-art algorithms. GoExplore is highlighted in yellow.

The purpose of this thesis is to understand why this discrepancy exists, by exploring how GoExplore differs from other game-playing algorithms. We will do so by analyzing game-playing algorithms from a tree search perspective, where I will argue that most algorithms perform a shallow and wide search of the game tree, while GoExplore drives a deep and narrow tree search. Concretely, this thesis makes the following contributions:

- It contrasts GoExplore with common Type A algorithms and discusses by which principles it successfully drives a deep and narrow tree search.
- It provides a practical contribution by improving the algorithmic time complexity of GoExplore.

2. *Tree Search Reduction Strategies*

Consider the game tree shown in Figure 2. Nodes define game states, and edges define actions the player can take. Leaf nodes are known as terminal nodes. Terminal nodes indicate that the game has ended and contain a value corresponding to the outcome when following that particular path.

We are currently in the root node, and we want to figure out which action we should take. From a bird's-eye view we see that the best outcome is at the green leaf node with value 3, so we should take the action leading to that node. But from the root node's perspective, we do not know how to get there. We only know which actions we can perform. Which action should we take?

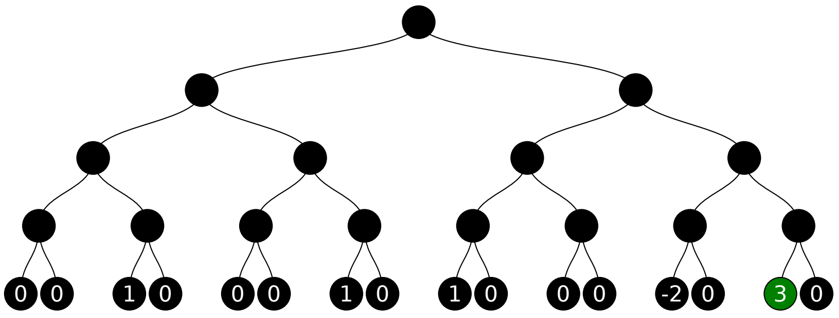


Figure 2: **Game tree.** Nodes are game states and edges are possible actions. Numbers at leaf nodes show the outcome of following following a trajectory. The green node highlights the best outcome.

One approach would be to back up the outcomes by recursively computing the maximum value from searching ahead one step, for both actions, until we reach the terminal nodes. Figure 3 shows the resulting backup tree. The values correspond to the values of the optimal value function, and the policy that greedily chooses the action that leads to the highest-value successor node, highlighted in green, is known as the optimal policy (see "**Appendix: MDP**"). We select the optimal action from any game state by following the optimal policy. From our root node, we do so by looking ahead one step and selecting the action that leads to the highest-value successor node. We repeat this decision-making process until we reach the terminal node.

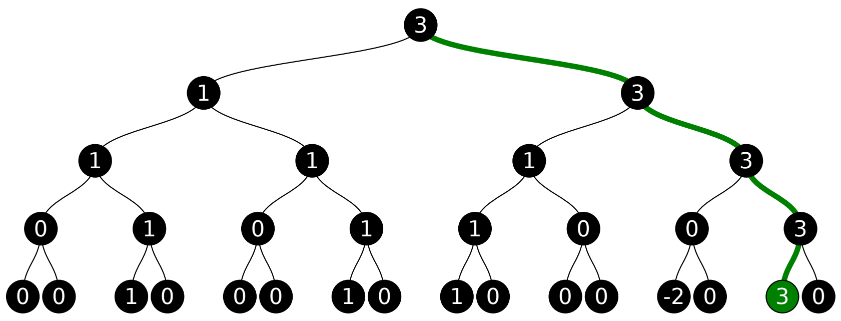


Figure 3: **Game tree after backing up values.** The highlighted path shows the optimal trajectory. The numbers are the values of the optimal value function.

This is a feasible strategy for small trees such as the one in Figure 2, but most game trees are far larger. For example, consider chess: on average you can make about 35 different moves from each position, and games end after roughly 80 moves [Russell and Norvig, 2010]. This leads to 35^{80} possible leaf nodes, which is an intractable size for any search algorithm.

The search can be narrowed down by creating a smaller approximation of the tree, either via depth reduction or breadth reduction. Claude Shannon referred to these two approaches as Type A and Type B strategies in his seminal paper on computer chess [Shannon, 1988]. GoExplore differs from most game-playing algorithms by following the Type B strategy.

We will first examine how some popular game-playing algorithms follow the Type A strategy of reducing depth and evaluating positions via value functions, and then we will look at how GoExplore drives a deep and narrow search of the game tree.

2.1. Type A - Tree Search by Depth Reduction

We can reduce the depth of the search by cutting the tree off at some depth and treating the new leaf nodes as terminal nodes. A terminal node is defined by having some value corresponding to the outcome of the game. Given a function that estimates the true outcome of reaching the new terminal node, we can follow the same procedure we did in the previous section to determine which action to take next: we traverse down to the terminal nodes and back up their values until we reach the root's successor states, then we choose the action leading to the highest-value successor state. We will first take a look at how some popular game-playing algorithms follow this approach.

Minimax Search using Offline Function Approximation. Minimax is a popular decision-making algorithm in computer chess [Russell and Norvig, 2010]. It follows a similar approach to the algorithm explained in the previous section, but it has to be adjusted slightly since we now have an opponent that will try to prevent us from following our optimal path.

Let us refer to the players as `MIN` and `MAX`. A negative outcome means that `MIN` won the game, and a positive outcome means that `MAX` won. Rather than always backing up the maximum value of the successor nodes, we now alternate between backing up the minimum and maximum value every other step. This will result in an optimal policy for each player: `MAX` always chooses the action leading to the highest-value successor node, and `MIN` always chooses the action leading to the lowest-value successor node.

The game tree for chess is far too large to search all the way to the terminal nodes, so we cut off the tree at some depth and treat the new leaf nodes as terminal nodes by evaluating their positions using some approximate value function (that approximates the optimal value function). This is known as heuristic minimax search. A common value function is the weighted sum of some features, where features could be, for example, "Number of white rooks" or "Has castled", etc. Using these values instead of the true outcome, we search and back up values as usual and select the action that leads to the highest-value successor state. Figure 4 shows an example with an inaccurate value function. We can find the best policy under this value function by backing up the minimax values, but it is far from optimal since the value function is a poor approximation of the optimal value function.

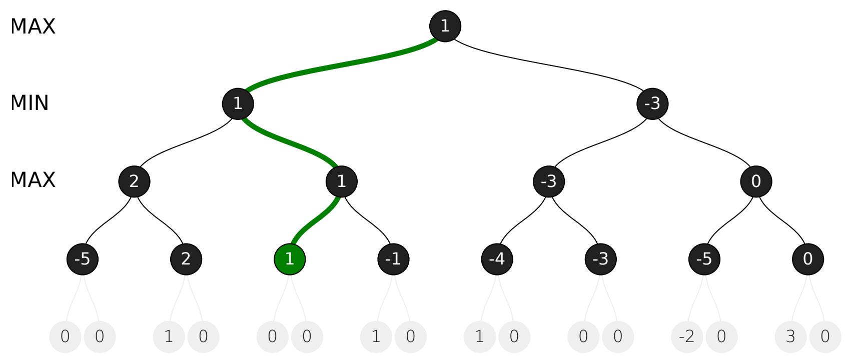


Figure 4: **Heuristic minimax tree**. Grey nodes are terminal states, which we assume are unreachable using exhaustive search. Values correspond to backed up minimax values. The highlighted path shows the optimal trajectory given the approximated value function.

Why do we not just evaluate the values of the successor states directly? This way we could skip the search altogether. It turns out that the value function more accurately approximates the true terminal values the deeper you search [Sutton and Barto, 2018], which is crucial for performance.

Monte Carlo Tree Search using Online Function Approximation. Monte Carlo Tree Search (MCTS) [Coulom, 2007] differs by estimating the value function for relevant states at decision time, rather the pre-computing the full function approximation offline. First, we initialize the search tree with just the root node, and then repeat the following steps for N simulations:

1. Selection: Starting at the root node, we traverse the search tree by following a selection policy until we reach a leaf node. The selection policy uses an exploration coefficient to balance exploitation and exploration. We want to search more efficiently by traversing the most promising paths we have encountered so far, but we also want to explore enough such that we can find nodes that potentially could have even higher values. A common selection policy is to take the action that leads to the node that maximizes the UCB1 score: $UCB1(s_i) = \frac{x_i}{n_i} + \alpha\sqrt{\frac{\ln N}{n_i}}$. The i:th node is denoted by s_i. It has a score and a visit count, denoted by x_i and n_i. N is the number of simulations, and α is the exploration coefficient.

2. Expansion: We grow the search tree by expanding the leaf node, i.e. we add its children to the frontier.

3. Simulation: We play out the game using a simulation policy until we reach a terminal state. A simple simulation policy could be to take random actions.

4. Backup: For each node that was traversed in the selection phase, we add the outcome of the simulation to the node's score and we increment the node's visit count.

Figure 5 shows an example of MCTS with 10 simulations. MCTS reduces depth in a different way: rather than cutting the tree at some depth, it starts at the root node and adds a new node at every iteration. This leads to shallow and wide trees as long as the exploration coefficient is non-zero. Note that we end up taking the sub-optimal action since the estimated values are poor approximations of the optimal values.

We could just perform simulations directly from the successor states – which is known as simple Monte Carlo Search – but, again, we end up with more accurate estimates if we build out a bigger tree [Russell and Norvig, 2010].

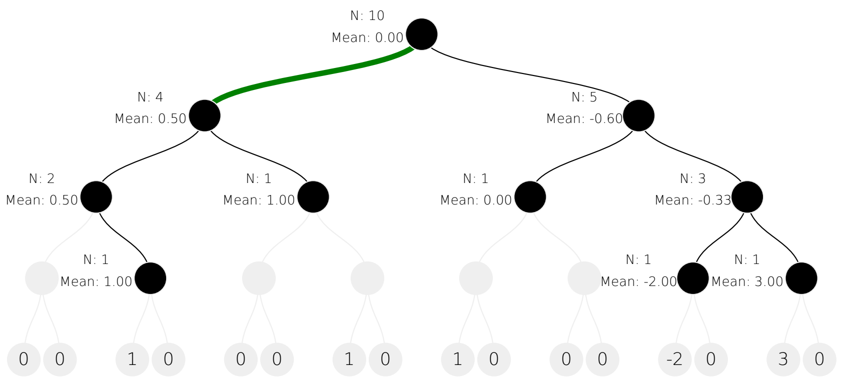


Figure 5: **MCTS tree after 10 simulations**. Grey nodes are terminal states, which we assume are unreachable using exhaustive search. Nodes that have been simulated from are shown in black. The numbers N and Mean correspond to the number of simulations involving a node and the average outcome, respectively. The highlighted edge shows the estimated best action.

2.2. *Type B - Tree Search by Breadth Reduction*

Another approach to making large tree search tractable is to narrow down the search, and therefore driving deep search. It is not clear how to successfully guide this type of search, so we will take a look at how GoExplore does it.

SPARSE REWARD ENVIRONMENTS. First we need to expand on the definition of the game trees we have discussed so far. GoExplore is run on the Atari Learning Environment (ALE) [Bellemare et al., 2013], where the outcome of a game is decomposed into intermediate rewards. After each action, we receive some reward. The reward can be zero, so we can think of the earlier game trees as implicitly having rewards of zero everywhere except for when we reach terminal states.

A particularly challenging game in the ALE is Montezuma's Revenge. It is a game where the player moves between levels by gathering items and opening doors while avoiding different types of obstacles. See Figure 6 for an example game frame. Few Type A strategies can outperform an average human, and many struggle to find even a single reward. GoExplore does not only surpass the average human, it even outclasses the human world record by over an order of magnitude [Ecoffet et al., 2019].

Type A strategies struggle with Montezuma's Revenge due to its sparse reward setting: even though intermediate rewards are provided, most of the time we will encounter rewards of zero. Recall that Type A strategies require a good value function to make good decisions. When we approximate the optimal value function, we rely on iteratively backing up some sort of reward signal. We end up with a poor approximation of the optimal value function if the reward signal is mostly "nothing at all", which results in poor decision-making.

Why do Type A algorithms not struggle with a game like chess, where non-zero rewards are only present at the end of the game? This is because Montezuma's Revenge still requires an even longer correct sequence of actions between rewards. A game of chess on average ends after roughly 80 moves [Russell and Norvig, 2010], while the shortest path to the first reward in Montezuma's Revenge requires a sequence of about 110 actions[1].

[1] This is an empirical measurement from playing the game myself.

GoExplore. The key idea behind GoExplore [Ecoffet et al., 2019] is to store encountered states in a table (the "archive"), and then exploring from the most novel among these states in the hope of finding new novel states. Novelty could be measured by, for example, how many times the state has been visited.

A state in the ALE is defined by the pixels of the game screen. We will not be able to add every single state to the archive due to memory constraints, so we need to reduce the state space somehow. A reduced state space also speeds up the search since we do not waste time exploring from too similar-looking states. State space reduction can be done by downsampling game frames into cells. This is done by (1) converting the image to grayscale, (2) reducing the resolution of the image, and (3) reducing the color depth of the pixels. Figures 6 and 7 show an example of a state and its corresponding cell.

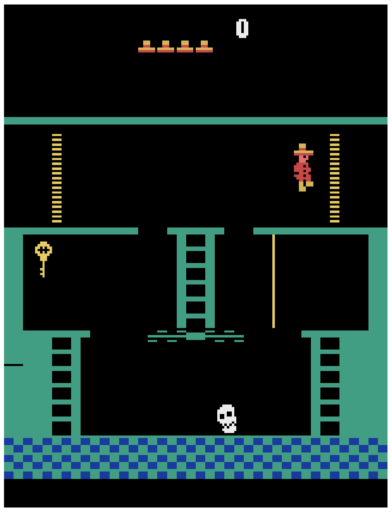


Figure 6: A game frame from Montezuma's Revenge.

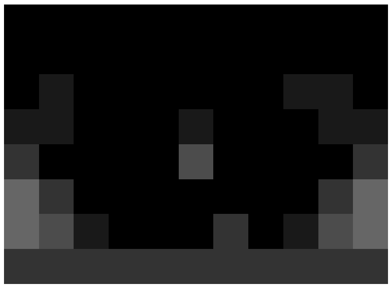


Figure 7: The corresponding 8x11 cell with 8 possible colors.

GoExplore initializes the archive with the starting cell, and then repeats the following steps for some number of maximum training frames:

1. Sample a promising cell from the archive by selecting according to cell novelty (see "3.2 Cell Selection"). Reset the simulator to the cell's game state.
2. Explore randomly for 100 steps. Each step, downsample the game state we receive from the environment into a cell. The archive will be modified if one of the following events occur:
   - **Cell discovery**: If a new cell is discovered, add it to the archive. The cell should store the game state, the trajectory to get to the game state, the cumulative reward, and the length of the trajectory.
   - **Cell update**: If a better cell is found, update the archive by replacing the old cell with the better cell. A cell is considered better if it has a higher cumulative reward so far, or if it has the same cumulative reward but a shorter trajectory.

In contrast to the Type A strategies discussed in the previous section, GoExplore does not compute value functions or policies. Instead, the algorithm outputs the trajectory to the highest-scoring cell. In this context, since the environment is deterministic, the trajectory is the sequence of actions from the starting state to the cell's game state.

How does this relate to game trees? A cell maps to a game state, so we could view a cell as a node on the game tree. Note that a cell will correspond to multiple nodes since similar-looking states map to the same cell. The archive is the set of cells we can explore from, which we can view as the frontier of the search. The frontier usually consists of leaf nodes, but in this context the frontier will also contain nodes in the interior of the tree. A cell can map to multiple nodes, so the frontier will, for each cell, contain the node corresponding to the highest-scoring version of the cell encountered so far. Figure 8 shows an example of a game tree after two 3-step explorations.

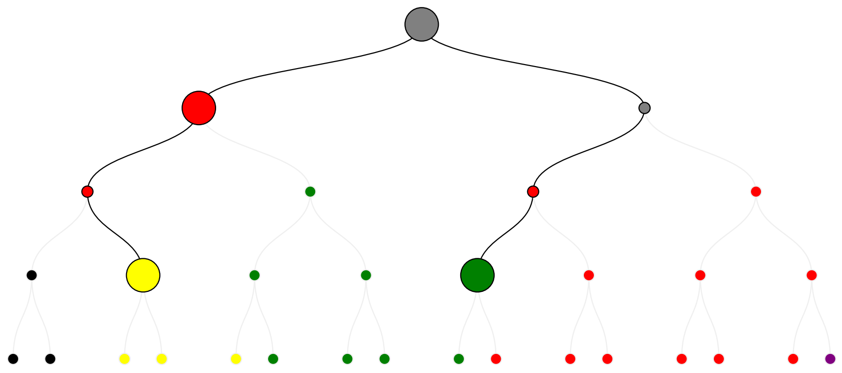


Figure 8: **GoExplore game tree**. Different colors indicate different cells. Nodes denote game states. The frontier consists of the nodes highlighted by the large circles.

Expanding and Re-Focusing the Frontier. How does GoExplore successfully drive deep search, particularly in sparse reward environments? Let us try to imagine the game tree near the start of the algorithm. Initially, we expand out in many different directions by performing 100-step random exploration runs. When we discover new cells, we add their corresponding node to the frontier. At this point, most of the frontier will be clustered near the root, see Figure 9a.

At some point, we will encounter our first reward. How will this affect the search tree? Let us say that we start one of our 100-step exploration runs, and after 5 steps we find a reward. The current cell will then have the highest score in the frontier, meaning that any cell encountered in the remaining 95 steps, whether old or new, will be moved or added onto this trajectory. Consequently, a large part of the frontier will be focused near the reward, and so we will be more likely to start from a cell near the reward next time we perform an exploration run. This will result in even more cells being moved near the reward, until the reward node effectively acts as the new "root" node of the tree, see Figure 9c. In other words, rewards re-focus the frontier near the reward node.

This mechanism of expanding and re-focusing the frontier will guide the frontier deeper and deeper as the search progresses. Figure 9d shows how most of the cells have clustered near the terminal nodes, at which point the algorithm is likely to find at least one terminal node.

We can draw an analogy to saving and loading from checkpoints when playing a video game. It is usually too difficult to beat a game from start to finish in a single run, so it is a common strategy to save the game near some challenging key events (for example, a boss fight). When we have progressed past one checkpoint, the remainder of the game is then clearly easier than the full game. This approach of progressing by moving checkpoints closer and closer to the end of the game is in essence how GoExplore guides deep and narrow search.

GoExplore does this by branching out the tree in many different directions when exploring and then pruning away branches when encountering a reward. Recall that cells are also updated even if they do not encounter rewards, as long as their trajectories are shorter. This can be seen as another type of (soft) pruning: if we find a cell with a short trajectory, we will not be able to add any worse version of the cell to the frontier. This idea is illustrated in Figure 10. If we end up finding the best red cell in the left sub-tree, the cluster of red cells in the right sub-tree will never be added to the frontier. We will then be less likely to explore that part of the tree since our only chance of exploring it is by starting from the root node. Note that the right sub-tree is not really being pruned: we can still explore it deeper if we manage to add the purple cell to the frontier. We are just more likely to explore the left sub-tree since that is where the frontier is mostly going to be located.

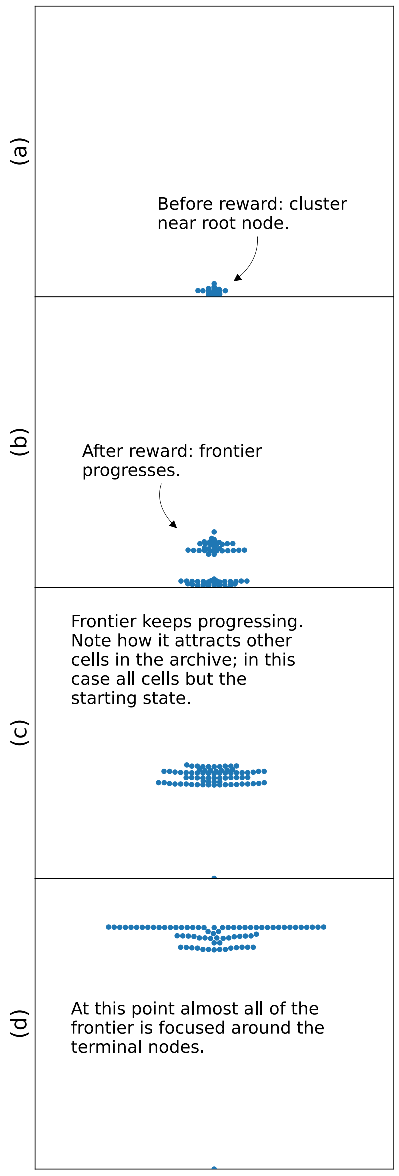


Figure 9: **Frontier progressing through the game tree.** These plots show the distribution of cell depths at different points of the search when playing Pong.

Finding New Cells by Guided Cell Selection. The search is mostly driven by cell discoveries since we rarely encounter rewards in sparse reward environments. We could increase the likelihood of cell discoveries by simply downsampling less, but then we risk wasting time exploring from states (and finding states) that are not meaningfully different.

A better idea is to start exploring from novel cells and hoping that we will discover new cells this way. Novelty is defined as a weighted sum of cell novelty and node novelty, measured by how many times a cell has been visited and how many times a node has been explored from, respectively. Note that the number of times a node has been expanded is reset after a cell update since the nodes will be different

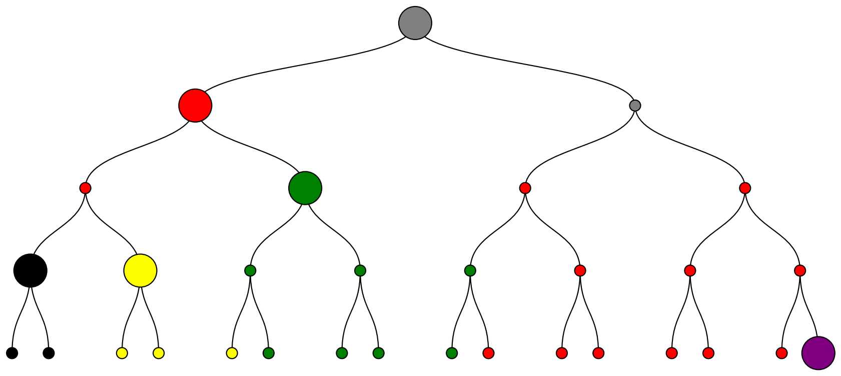


Figure 10: **GoExplore shortest trajectories.** The cells with the shortest trajectories are highlighted by large circles.

even if they correspond to the same cell.

I run the algorithm over 1 billion game frames on Montezuma's Revenge using guided[2] versus random cell selection to investigate whether guided selection increases the likelihood of finding new cells. The results are shown in Figure 11. Both strategies quickly discover many cells corresponding to the early levels of the game, but the discoveries diminish until we reach about 500 million frames. At that point guided selection finds a cluster of new cells, likely corresponding to new levels of the game. Random selection, however, does not manage to find this cluster of cells even after 1 billion frames. This supports the idea that by starting exploration from novel states, we are more likely to find other novel states (even with random exploration).

[2] Stochastic acceptance selection, see section "3.2 Cell Selection".

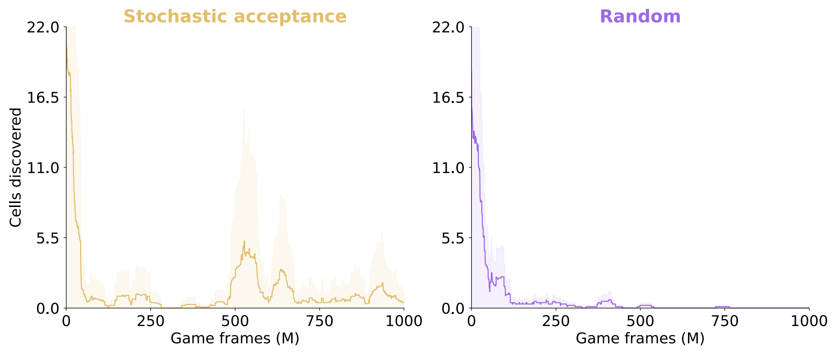


Figure 11: **Guided selection enhances cell discovery**. The left graph shows the number of cell discoveries per iteration using guided (stochastic acceptance) cell selection, and the right graph shows the same quantity using random cell selection.

Further Work. Given this mental framework of understanding Go-Explore as a systemic way of expanding and re-focusing the frontier, further research could explore whether more aggressive re-focusing increases the speed of the algorithm.

One idea is to use some mechanism by which we automatically set a reasonable “search speed” (which draws some inspiration from learning rate schedules in deep learning). One could start with a large "step size" by using a large downsampling factor such that the cell space will be small. Then we search in this cell space until we have not found any new or better cells for a while. When the search has saturated, we can decrease the step size by doing a slightly more fine-grained downsampling, which defines a slightly larger cell space. This way we exploit the faster search speed afforded by small cell spaces, and if we get stuck, we make the cell space larger. This would also remove the need to tune the downsampling parameters manually.

We could also tune how strongly we re-focus the frontier on rewards. One idea is to use "hard" checkpoints – every time we find a reward, we reset the archive and act as if the reward checkpoint is the new starting state. This way we commit completely to the first promising path we encounter. We would risk getting stuck in suboptimal trajectories, but if the solution is "good enough", the benefits of the increased search speed might outweigh the costs.

Another way to attract more cells to the reward cell, without removing all old cells, would be to explore longer following a reward. For example, if we find a reward, we could explore for 10000 more steps rather than the standard 100 steps. This way we will encounter more cells which will be moved near the reward cell. Another approach would be to do a full-width forward search following a reward. So let us say we are at step 60 in our exploration run, and we find a reward. Rather than just performing random actions for the remaining 40 steps, we could instead perform a full-width search from this point, for however long we can afford to.

3. *Algorithmic time complexity*

The original GoExplore implementation runs in superlinear time, meaning that an exploration run takes longer as the algorithm progresses. This is mainly because cell selection scales linearly with the number of cells, but also because trajectory tracking becomes more costly as the trajectories grow larger. In this section, we will discuss how to make GoExplore’s runtime scale linearly with the number of exploration runs, by achieving constant-time trajectory tracking and constant-time cell selection.

3.1 Maintaining trajectories

GoExplore outputs the trajectory to the highest-scoring cell, so we need to make sure each cell tracks the trajectory needed to get to it from the start state. We can do this by storing the trajectory when a cell is found or updated, and by loading its trajectory when a cell is selected. The naïve approach is to store the full trajectory from start to finish for every cell. Consequently, both storing and loading will have a time complexity of $O(|\tau^*|)$, where τ^* denotes the longest trajectory for any cell in the archive. This number will grow as GoExplore progresses.

The main problem is that there is major overlap between stored trajectories. Consider, for example, that we first find a cell A and then a cell B in the same exploration run. Cell A will contain the actions needed to get to A, and cell B will contain the actions needed to get to B *plus* the actions needed to get to A.

A better approach is to only store the sequence of actions from the previous cell as well as a backward pointer to that cell. Then we can follow the backward links until we reach the start state, while concatenating all the trajectory partitions found along the way into the full trajectory. This way trajectory loading and storing will take at most $O(100)$ time since an exploration run consists of 100 timesteps. However, we can not link between cells directly due to cycles; if we find cell A, then B, and finally a higher scoring A, we will end up with a cycle between A and B. To resolve this, we instead link between the unique nodes of the game tree and let the cells link to those nodes, as illustrated in Figure 12.

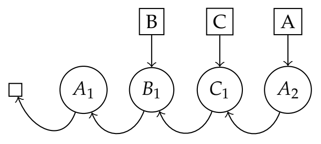


Figure 12: **Linked trajectory partitions after cell update.** Nodes corresponding to any discovered or updated cells are denoted by circles. They contain a pointer to the previous node, and an array of the actions needed to get from the previous node to the current node. Cells, denoted by squares, contain a pointer to the highest-scoring corresponding node. Cell A initially pointed to node A_1, but after finding node A_2, which maps to cell A and has a higher score than A_1, cell A gets updated and now points to A_2.

A third approach, which achieves $O(1)$ loading and storing, is to store every single action with backward links. This turns out to be impractical due to excessive memory usage. In my main experiment setup – 1 billion game frames on Montezuma's Revenge, resulting in roughly 20 thousand cells – the second and third approaches resulted in roughly 1GB and 60GB of memory usage, respectively.

3.2 Cell Selection

Let us analyze the time complexity of cell selection. We maintain an array of cells and an array of their corresponding novelty weights. A weight for a cell c is defined as $w_1\sqrt{\frac{1}{c_{visits}}} + w_2\sqrt{\frac{1}{c_{selected}}}$, where c_{visits} and $c_{selected}$ refers to the number of times the cell has been visited and selected, respectively, and w_1, w_2 are hyperparameters.

We want to select a cell from the array of cells probabilistically according to the corresponding weights. Roulette wheel selection (see Figure 13) is the standard solution to this problem [Lipowski and Lipowska, 2012]. It consists of two steps:

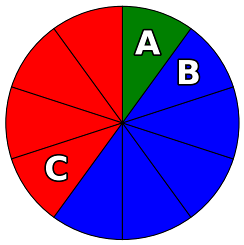


Figure 13: **Roulette wheel selection**. Roulette wheel selection involving three cells A, B, and C, with selection probabilities 10%, 50%, 40%, respectively. Each cell gets assigned a fraction of the wheel according to its selection probability. The selection consists of picking a random spot on the wheel and then returning the cell associated with that spot.

1. Construct the roulette wheel. First, we normalize the weights into probabilities by computing the sum of all elements and dividing every weight by this sum. Then we create the wheel by computing

the cumulative sum of the probabilities for each cell – this way the cells get assigned a fraction of the wheel proportionate to their weights.

2. Spin the wheel. We generate a random number between 0 and 1 and perform binary search to find out in which fraction of the wheel the random number ends up. We return the corresponding cell.

Let n denote the number of cells in the archive. Then the operations take $O(n+n+n+n)$[3] and $O(1+\log n)$[4] time, respectively, resulting in a total time complexity of $O(n)$. The first operation would only need to be computed once if we had a static array of weights, but in our setting, we need to modify the array after every iteration. This will lead to superlinear time complexity for GoExplore if the archive grows as the algorithm progresses, which is commonly the case.

[3] For computing the sum, copying the array, dividing elementwise, and computing the cumulative sum.

[4] For generating a random number, and performing binary search.

We would like GoExplore's time complexity to be independent of the size of the archive, which requires constant-time cell selection. This is achievable using stochastic acceptance selection [Lipowski and Lipowska, 2012], which consists of the following steps:

1. Randomly pick a cell from our archive.

2. Compute an acceptance probability $p_{accept} = w_i/w_{max}$, where w_i is the cell's weight and w_{max} denotes that maximum weight in the archive.

3. Generate a random number $p \in [0,1]$.

4. If $p < p_{accept}$, return the cell. Else, go back to (1).

I use a maximum bound $U \geq w_{max}$ rather than w_{max} in the implementation, which can still lead to $O(1)$ cell selection in practice [Lipowski and Lipowska, 2012].

To measure what practical impact stochastic acceptance selection has on the total runtime, I run GoExplore with both selection algorithms over an increasing number of game frames. As the number of game frames increases, so does the size of the archive. Figure 14 shows the results. We see that roulette wheel selection takes longer as the archive size grows, while stochastic acceptance selection leads to a linear total runtime. For further context on the practical implications: in my main experiments, GoExplore processed 1 billion game frames in roughly 2 days using stochastic acceptance selection, and in slightly less than 3 days using roulette wheel selection.

The results show that the time complexity is independent of the archive size when we select by stochastic acceptance, but how is GoExplore's performance affected? I measure this by comparing the

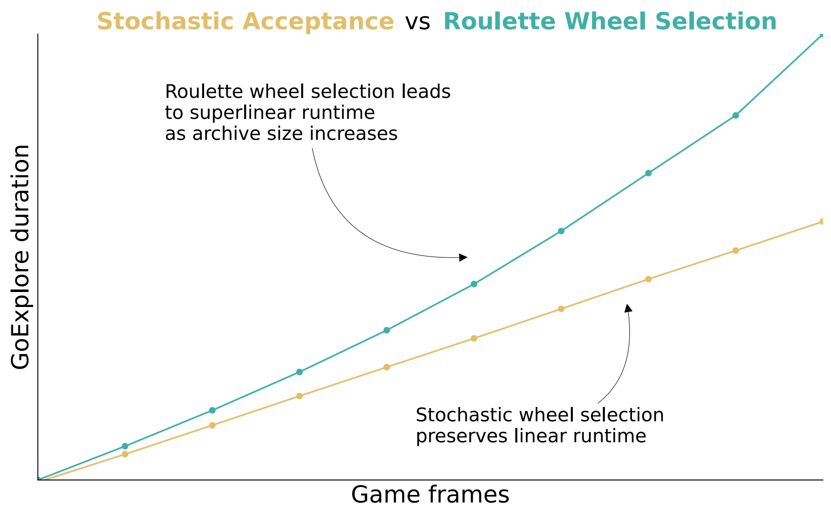


Figure 14: **Stochastic acceptance selection preserves linear total runtime despite increasing archive size**. The figure compares the runtime of GoExplore using stochastic acceptance vs roulette wheel selection, sampled over a varying number of game frames.

raw scores on Montezuma's Revenge over 1 billion game frames. Table 1 summarizes the results. For further context, I also include the performance from random cell selection, as well as the current state-of-the-art results achieved by GoExplore [Ecoffet et al., 2019] and Agent57 [Badia et al., 2020]. There is a large variance in the scores, but stochastic acceptance performs at least as well as roulette wheel selection, likely even better. The original GoExplore algorithm by Ecoffet et al. [2019], using roulette wheel selection, substantially outperforms this implementation. This can partially be explained by the fact that no hyperparameter tuning is done during these experiments (since they are too costly). Still, stochastic acceptance selection leads to better performance than Agent57, which is the current state of the art excluding any GoExplore variants.

Algorithm	Mean	95% CI
Random	6,900	6,700-7,100
Roulette	12,225	7,650-18,900
StochAcc	22,560	12,980-32,140
Ecoffet	57,439	47,843-67,224
Agent57	9,352	6,413-12,291

Table 1: Scores on Montezuma's Revenge.

4. *Discussion*

A key limitation with GoExplore is that it only produces a trajectory from the start state to the goal state. Any deviation from this trajectory would render the algorithm useless. In most games, we will almost certainly diverge from our optimal path; either due to stochasticity in the environment, or by facing an opponent that will try to prevent us from reaching the goal state.

GoExplore tackles this problem by also learning a policy (see "**Appendix: MDP**"). There exist two variations:

- **"Robustified" GoExplore:** Use the extracted trajectory as a demonstration to learn a policy [Ecoffet et al., 2019, Salimans and Chen, 2018]. We do this by starting near the end of the trajectory, and then interacting with the environment until the policy reliably can reach the goal state. The agent is then pushed backwards some number of steps, and then it interacts until it reaches the goal state again. This process is repeated until the agent is pushed back all the way to the start state, at which point the policy can beat the full game.

- **Policy-based GoExplore:** Learn a goal-conditioned policy that returns to a selected cell[5], and then explores from that point on [Ecoffet et al., 2021].

[5] Rather than using a simulator.

Both these approaches have been tested on Montezuma's Revenge with stochasticity injected[6], and they have been shown to still outperform human experts [Ecoffet et al., 2019, 2021]. A key insight with this thesis is that only a very narrow part of the state-space is explored during learning. To further test the generality of the policy-based agents, I suggest further exploring settings where the agent mainly acts outside the narrow state-space it has been trained on. For instance, one could use the simulator to change the starting state of the game. The agent will then start in a completely different part of the search tree – will it still be able perform well? The ultimate test would be a multiplayer game, where the opponent will not let the agent come anywhere near the sub-space that our agent has been trained on. Can a GoExplore policy perform well in a turn-based game like chess, for example?

[6] See "**Appendix: Method and Experiment Details**".

Another direction for future reasearch could be to test GoExplore on classical search problems since GoExplore is, in essence, an approximate search algorithm.

5. *Conclusion*

This thesis shows how GoExplore differs from common game-playing algorithms by driving a deep and narrow search of the game tree. GoExplore progresses the frontier by:

- expanding the frontier via cell discoveries in absence of a reward signal,
- re-focusing the frontier via cell updates following a reward signal,
- and systematically exploring novel parts of the state space using guided cell selection.

Furthermore, this thesis provides an open-source implementation of GoExplore. It does not perform as well as the original implementation by Ecoffet et al. [2019], likely because no hyperparameter tuning is done, but the algorithmic time complexity has been improved to linear-time. It achieves linear time complexity by only storing intermediate trajectory partitions between cells rather than the full trajectories, and by doing constant-time cell selection using the stochastic acceptance method. Stochastic acceptance selection does not only lead to faster runtime, but it also performs at least as well as standard roulette wheel selection.

References

Adrià Puigdomènech Badia, Bilal Piot, Steven Kapturowski, Pablo Sprechmann, Alex Vitvitskyi, Zhaohan Daniel Guo, and Charles Blundell. Agent57: Outperforming the Atari human benchmark. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 507–517. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/badia20a.html.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013. DOI: 10.1613/jair.3912. URL https://doi.org/10.1613%2Fjair.3912.

Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers, editors, *Computers and Games*, pages 72–83, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-75538-8.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems, 2019. URL https://arxiv.org/abs/1901.10995.

Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O. Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, Feb 2021. ISSN 1476-4687. DOI: 10.1038/s41586-020-03157-9. URL https://doi.org/10.1038/s41586-020-03157-9.

Bradley Efron. Nonparametric standard errors and confidence intervals. *The Canadian Journal of Statistics / La Revue Canadienne de Statistique*, 9(2):139–158, 1981. ISSN 03195724. URL http://www.jstor.org/stable/3314608.

Yijie Guo, Jongwook Choi, Marcin Moczulski, Shengyu Feng, Samy Bengio, Mohammad Norouzi, and Honglak Lee. Memory based trajectory-conditioned policies for learning from sparse rewards. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 4333–4345. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/2df45244f09369e16ea3f9117ca45157-Paper.pdf.

Adam Lipowski and Dorota Lipowska. Roulette-wheel selection via stochastic acceptance. *Physica A: Statistical Mechanics and its Applications*, 391(6):2193–2196, 2012. DOI:

10.1016/j.physa.2011.12.0. URL https://ideas.repec.org/a/eee/phsmap/v391y2012i6p2193-2196.html.

Marlos C. Machado, Marc G. Bellemare, Erik Talvitie, Joel Veness, Matthew Hausknecht, and Michael Bowling. Revisiting the arcade learning environment: Evaluation protocols and open problems for general agents. *J. Artif. Int. Res.*, 61(1):523–562, jan 2018. ISSN 1076-9757.

Sebastian Raschka. Model evaluation, model selection, and algorithm selection in machine learning, 2018. URL https://arxiv.org/abs/1811.12808.

Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

Tim Salimans and Richard Chen. Learning montezuma's revenge from a single demonstration, 2018. URL https://arxiv.org/abs/1812.03381.

C. E. Shannon. *Programming a Computer for Playing Chess*, page 2–13. Springer-Verlag, Berlin, Heidelberg, 1988. ISBN 0387913319.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. Adaptive Computation and Machine Learning. MIT Press, Cambridge, MA, 2 edition, 2018. ISBN 978-0-262-03924-6. URL http://incompleteideas.net/book/the-book.html.

Appendix: Method and Experiment Details

This thesis follows the methodology outlined in the original GoExplore paper [Ecoffet et al., 2019]: *sticky actions* are not used, and a constant ($k = 4$) *frame skipping* is applied to the environment. Sticky actions means that we inject some stochasticity into the environment by sometimes ignoring the agent's current action, and instead reusing the previous one [Machado et al., 2018]. This is incompatible with simulator-based GoExplore, since any stochasticity would break the stored trajectories. Frame skipping makes the agent only perform a decision every k steps. It is applied since it is empirically known to speed up learning [Machado et al., 2018]. To be consistent with the original paper, all reported frames refer to *game frames*, i.e. the number of *training frames* times the frame skip constant k. To make the distinction clear: 1 billion game frames means that the agent interacted with the environment 250 million times, i.e. it processed 250 million training frames.

Furthermore, the exploration is not really random. The agent uses *action repetition*, meaning that it will repeat its previous action with 95% probability[7] (and otherwise take a random action). The motivation behind this is to explore in a consistent direction, which is particularly helpful on Montezuma's Revenge. An ablation study by Ecoffet et al. [2021] shows that action repetition significantly improves performance on Montezuma's Revenge, but they still manage to beat the state-of-the-art even with random exploration.

[7] This is different from sticky actions, where the *environment* ignores the agent's decision.

GoExplore uses five hyperparameters in total: (1) the width w, (2) height h, and (3) color depth d of the downscaled frame, as well as (4) a cell novelty weight w_1 and (5) a node novelty weight w_2. The reported experiments use the following hyperparameters[8]: $w = 8$, $h = 11$, $d = 8$, $w_1 = 0.1$, $w_2 = 0.3$.

[8] This matches the hyperparameters used by Ecoffet et al. [2019].

To measure uncertainty, 95% bootstrap confidence intervals are computed using the percentile method [Raschka, 2018, Efron, 1981]:

1. Produce a sample of n runs with varying random seeds.
2. Compute a bootstrap distribution. Create a bootstrap sample by sampling with replacement n times, and then calculate the mean of that sample. Repeat this b times, to get a distribution with b samples.
3. Define the lower bound, the mean, and upper bound, as the 2.5th, 50th, and 97.5th percentile of the bootstrap distribution.

The main experiments were run on Montezuma's Revenge for 1 billion game frames. These were costly, so n was at most 5-10 for any single experiment. All confidence intervals were computed by resampling $b = 10000$ times. All experiments were run on the Google

Cloud Engine, using their Intel Cascade Lake CPU platform with 2 cores and 4GB of RAM.

Appendix: MDP

A sequential decision problem, such as a game, can be formalized as a Markov Decision Process (MDP). The decision-maker is known as an *agent*, and it performs decisions inside an *environment*. The agent performs an action a_t in state s_t at timestep t, and receives a reward r_{t+1} and a new state s_{t+1} from the environment.

Formally, a MDP is defined by:

- S, a set of possible states,
- $A(s)$, a set of possible actions from any state s,
- $T(s'|s,a)$, a state transition model,
- and $R(s,a)$, a reward function.

An MDP can have varying levels of complexity. To make it relevant for the Atari Learning Environment [Bellemare et al., 2013], we make the following assumptions: the model is known and accurate; the environment is fully observable by the agent; the state and action spaces are discrete; and the process terminates after a finite number of decisions.

The solution to an MDP is the sequence of actions that maximize the expected sum of rewards (or *return*) $G = \sum_0^T r_i$. The expected return from any state s is given by a *value function* $V(s)$. The agent decides what actions to take by following a *policy* $\pi(s)$, which tells the agent what decision to make in any state s. The maximum expected return is found by following an *optimal policy* $\pi^*(s)$

Given that the transition model is known and accurate, the optimal policy can be found by exhaustively sweeping over the full game tree and backing up the maximum value of the successors for each node. This is generally memory inefficient and slow. A more efficient solution can be found using *dynamic programming*. In the context of MDPs, the algorithms iteratively compute more accurate value functions and policies. They initialize the value function and the policy, and then repeat some variation of the following steps until convergence:

1. Policy evaluation: Evaluate the value function under the current policy. This can be done by looking ahead one step under the current policy, and updating the current value to be the retrieved reward plus the expected value of the next state:

$$V_{k+1}^{\pi}(s) = R(s,\pi(s)) + \sum_{s'} T(s'|s,\pi(s))V_k^{\pi}(s').$$

2. Policy improvement: Improve the policy by acting greedily with respect to the value function. This can be done by maximizing the

policy evaluation equation:

$$\pi(s) = \arg\max_a \left(R(s,a) + \sum_{s'} T(s'|s,a)V(s') \right).$$

The Atari games using a simulator can be seen as *search problems* – a special case of MDPs where the transition function is deterministic. The methods mentioned above can still be used to find the optimal value function and policy. The simulator-based GoExplore can only be applied to search problems since any stochasticity in the environment would break the trajectories that are stored in each cell. There exist policy-based variations of GoExplore [Ecoffet et al., 2021, Guo et al., 2020], where the agent learns a policy that returns to cells and then explores, which can be applied to the more general case where the transition function is stochastic.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-06-07

EXAMENSARBETE Understanding Deep and Narrow Tree Search with GoExplore
STUDENT Patrik Persson
HANDLEDARE Volker Krueger (LTH)
EXAMINATOR Elin Anna Topp (LTH)

Hur GoExplore lär sig att spela Montezuma's Revenge

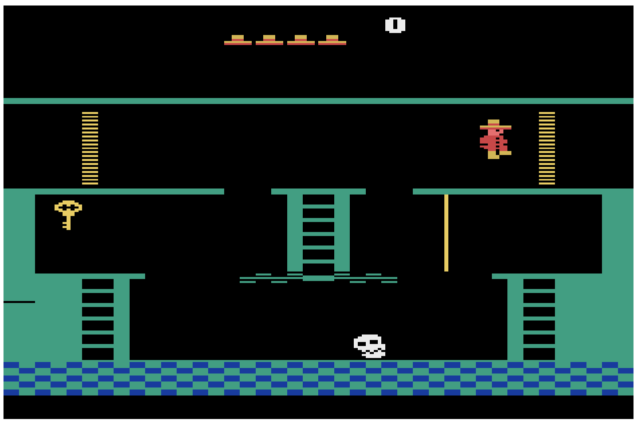
POPULÄRVETENSKAPLIG SAMMANFATTNING **Patrik Persson**

Till skillnad från andra AI-algoritmer så är GoExplore betydligt bättre än någon människa på att spela Montezuma's Revenge. Detta arbetet förklarar vad som gör GoExplores tillvägagångssätt unikt.

Det finns två generella strategier för att lära sig att spela spel: man kan antingen utforska spelet brett, eller djupt. När du lär dig spela schack kan du försöka lära dig alla de bästa öppningarna, vilket ger dig bred kunskap om hur du bör spela öppningsfasen. Nackdelen är att du inte kommer ha spelat mycket schack efter öppningen. Ett annat alternativ är att välja en öppning, och sen spela en massa matcher utifrån den, vilket kommer ge dig djup kunskap inom just den här öppningen.

De flesta AI-algoritmer för spel följer den första strategin, d.v.s. de försöker lära sig att spela spelet genom att få en bred och fullständig bild av hur det bör spelas. Detta fungerar väl för många spel, exempelvis brädspel som schack och Go, men även datorspel som Atari-spelen. Men just Montezuma's Revenge har visat sig vara en oerhört tuff utmaning; de bästa AI-algoritmerna kan inte slå en genomsnittlig människa, och många kan inte ens ta en enda poäng!

Grunden till inlärning, i det här sammanhanget, är att använda sig av belöningar och straff. Om AI:n gör något dåligt, exempelvis förlorar spelet, så blir den bestraffad. Om den gör något bra blir den belönad. Över tid lär den sig att undvika dåliga beteenden och söka bra beteenden. Svårigheten med Montezuma's Revenge är att man väldigt sällan blir belönad eller bestraffad, vilket leder till väldigt långsam inlärning.



GoExplore följer den andra strategin. Istället för att få en bred kunskap om hur spelet ska spelas så siktar den tidigt in sig på en lovande strategi, och fördjupar sig sedan i den. Detta kan kopplas till den tidigare nämnda liknelsen till schack: istället för att lära sig massa olika öppningar så väljer vi en öppning och utforskar den fullt ut.