Abstract

Recent advances in Game AI have mostly focused on two-player, zero-sum, deterministic games such as Chess and Go. While these environments have driven major breakthroughs, they do not reflect many of the challenges found in more complex, real-world decision-making settings. In particular, multi-player games with stochastic elements introduce a much larger state space, non-trivial opponent modeling, and unclear adversarial boundaries.

This paper introduces Deep Diver, a research testbed and AI agent built for *Deep Sea Adventure*, a Japanese board game defined by stochastic movement and a tightly coupled shared resource between 2 and 6 players. Unlike classic competitive games, all players in *Deep Sea Adventure* rely on a single oxygen supply, meaning that one player's greed directly harms everyone else.

The core idea is to adapt standard Monte Carlo Tree Search to better reflect the mechanics of *Deep Sea Adventure*. Traditional MCTS assumes deterministic state transitions. In contrast, this environment relies heavily on dice rolls.

Introduction

Motivation

I chose this project because *Deep Sea Adventure* sits at an interesting intersection of game theory, probability, and shared-resource management that is still underexplored in AI research. While many "push-your-luck" games exist, very few enforce such a strict, fully shared constraint.

The oxygen system creates a semi-cooperative dynamic early in the game that gradually turns into cutthroat competition. Solving this requires an agent that can manage risk at both the individual and group level, a challenge that closely mirrors real-world multi-agent systems operating under shared constraints.

Traditional "win–loss" assumptions begin to break down as soon as more than two players are involved. An agent must decide not only *how* to win, but *who* to oppose and *when*. This becomes even more difficult when players share a limited resource, creating a natural "Tragedy of the Commons" scenario. In *Deep Sea Adventure*, every move

changes the survival window for all players, forcing agents to balance individual gain against collective collapse.

Contributions

- **Game Environment**
  A complete C++ implementation of *Deep Sea Adventure*, used as a research testbed.

- **Heuristics agent**

- **Pure Monte Carlo Tree Search**

- **Single Threaded MCTS agent**

- **Multi Threaded MCTS agent, both with heuristics and with random rollouts**

Related Work

Relevant prior work includes:

- Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, Simon Colton (2012) - **"A survey of Monte Carlo tree search methods"**
  [https://research.monash.edu/en/publications/a-survey-of-monte-carlo-tree-search-methods/]

  This comprehensive survey of Monte Carlo Tree Search laid the foundation for the decision to use MCTS in complex game settings. It outlines the basic algorithm, key enhancements like UCT (Upper Confidence Bounds applied to Trees), and the broad variety of domains where MCTS has been applied. Reviewing this work helped clarify which modifications are "standard practice" versus which are novel in multi-player or stochastic contexts, and guided the choice to extend MCTS with chance nodes rather than redesign the whole search paradigm.

- Nicolas Jouandeau, Tristan Cazenave (2010) - **"Monte-Carlo Tree Reductions for Stochastic Games"** [http

s://www.lamsade.dauphine.fr/~cazenave/papers/mctrsg.pdf]

This work examines how Monte Carlo Tree Search can be adapted for stochastic games, where random events (chance outcomes) cause huge branching factors that standard MCTS struggles with. The authors propose using **c**hance-nodes, which explicitly represent random events like dice rolls, and other tree reduction strategies (e.g., move grouping) to manage branching explosion. They experiment with these ideas in games such as Chinese Dark Chess (a game with both strategic moves and random outcomes) to show how different reduction policies affect performance and tree structure. This paper is directly relevant to this project because it explores formal ways to incorporate chance nodes and reduce complexity in stochastic search trees, which informs how stochastic MCTS differs from classic MCTS.

- Sturtevant (2008) - **"An Analysis of UCT in Multi-Player Games"** [https://webdocs.cs.ualberta.ca/~nathanst/papers/multi-player_UCT.pdf]

  This paper analyzes how the UCT variant of Monte Carlo Tree Search behaves when extended to multi-player game settings. Traditional search methods such as *maxn* compute pure strategy equilibria, but UCT, by virtue of its stochastic sampling and exploration/exploitation trade-off, tends to compute mixed-strategy equilibria in multi-player trees. Sturtevant compares UCT's performance against conventional multi-player search algorithms across several domains (e.g., Chinese Checkers, Hearts, Spades) and shows that with a sufficient number of simulations, UCT performs competitively or better than established approaches. This work is directly relevant to this project because Deep Sea Adventure is an N-player environment with interdependent payoffs, and Sturtevant's analysis provides theoretical and empirical justification for using UCT-based MCTS in multi-agent decision spaces rather than relying on two-player or minimax assumptions.

- Schrittwieser et al. (2021) - **"Planning in Stochastic Environments with a Learned Model"** [https://iclr.cc/virtual/2022/poster/6832]

  This paper extends modern model-based planning approaches like MuZero to environments with inherent stochasticity. Traditional deep planning methods such as AlphaZero and MuZero assume deterministic transitions, but this work introduces Stochastic MuZero, a variant that learns a stochastic model of the environment and performs stochastic tree search with explicit representation of random transitions. While the focus is broader than board games, the core idea,

incorporating randomness into the planning model and handling it within the search process, is extremely relevant to this project's stochastic MCTS design. This research shows how planning algorithms can maintain performance in domains with random transitions by separating action effects and chance outcomes, much like this project's use of chance nodes to represent dice rolls within MCTS.

Deep Sea Adventure

[Gameplay information taken from the official manual of the game included in the game box]

The Mechanics of Deep Sea Adventure

Deep Sea Adventure is a multi-player push-your-luck race, divided in three rounds, consisting of N players (2-6) and a linear path of treasure chips.

The core mechanics are:

• **Shared Resource Constraint:**
All players share a single oxygen supply, initialized at 25 units. The oxygen depletes at the start of every turn based on the total number of treasures a player is currently holding. This creates a "Tragedy of the Commons" dynamic where one player's greed penalizes the entire group.

• **Stochastic Movement:**
Players move by rolling two 3-sided dice. The effective movement M is calculated as M = Roll(2d3) − Theld, where Theld is the number of treasures carried. This introduces a negative feedback loop: greedier players move slower and consume more oxygen.

• **The Goal:**
The objective is to maximize collected treasure points (P) across the three rounds, while returning to the submarine before the oxygen reaches zero. Failure to return results in P = 0 for the specific round.

Detailed explanation of the rules

- The chips get more valuable the deeper you go. There are 32 total treasure chips, split equally into 4 types. The road starts with the lowest value chips and continues with all of the categories one by one.
  Each category contains a range of 4 possible values that the chips can take, having two chips exactly of each value, as follows:

  Type 1 chips, triangle shaped: between 0 and 3 points.
  Type 2 chips, square shaped: between 4 and 7 points.
  Type 3 chips, pentagon shaped: between 8 and 11 points.
  Type 4 chips, hexagon shaped: between 12 and 15 points.

  As we can see above, there is a possibility that getting two type one chips could still result in score zero, or that having one type 2 chip (average value 5.5) is almost always better than having two type one chips (average value 3), but having two type two chips is slightly better than one type 2 chip (value only) and two type three chips are much better than one type 4 chip, getting a higher value for any combination. This range induces variability and uncertainty to the game, shifting the heuristics of the game.

- The spaces from which treasures have been taken are replaced temporarily with a blank chip. This works like a normal path chip except that it cannot be picked up by any player, having no value attached to it, and it can be instead used to ‘drop’ a treasure from the player’s current round collection, if they wish to do so. The dropped treasure chip will replace the blank one and the player will be able to continue moving easier, without knowing the exact value dropped.

- The value of the chips taken is hidden until the player arrives at the submarine. That changes the gameplay because the player can’t safely throw away treasure that is low in value, and doesn’t know whether a win is guaranteed due to lucky high value picks before finishing the round. It must all be calculated based on many probabilities.

- If a player with treasure picked up the current round doesn’t make it back to the submarine before the air runs out, the treasure sinks at the bottom of the ocean, which means that all fallen treasure is stacked in piles of maximum three chips on the last empty spots of the road. These stacks, although they have the value of all the components, only count as one treasure when it comes to oxygen

consumption and dice throw subtraction, which makes them generally the most valuable treasures, especially as chips keep stacking until the last round.

- At the end of every round, the blank chips are removed and the road shortened. This way, the last round makes it way easier to get a big amount of points and most players plan in advance for it. It might be the most decisive round of the three.

- Once you declare you head back, you cannot return towards the depths again. The return point must be carefully chosen, and usually once a player turns back the others tend to follow so that they won't drown in the race back.

- You can skip over other players when you reach an occupied chip, without subtracting that skip from the dice roll. That gives you a nice boost, especially if there are multiple players in front of you. This also balances the game when there are many players and can give an advantage to the players that are behind.

- The last player that got in the submarine will start the next round.

Tactics

This game has a lot of randomness in it, but despite that, all pieces present are known from the start of the game and so probabilities could be calculated for the optimal moveset. These calculations would get very complex quite fast, so it remains to test whether an artificial intelligence agent can find an improved playstyle.

As for humans, here are some tactics that seemed to be useful and could be implemented in an heuristic game bot, from my own experience, advice given by others or forum discussions:

[The forums:
https://opinionatedgamers.com/2015/01/01/deep-sea-adventure-first-impression/
https://boardgamegeek.com/blog/7177/blogpost/79372/deep-sea-adventure-strategy-tips]

- Turn back fast: going too deep is possibly the deadliest mistake of the game. The oxygen finishes quicker than expected, especially since players take as many

treasures as possible on their way back to the submarine. Going back after two or three rolls is usually a good strategy, even if you won't reach the deep treasures from the first rounds. The most treasure is usually saved for the last round.

- Following the first player who turns back is also good: they might plan to pick up a lot of treasure on their way back.

- Prepare for the third round: this is usually where the most points are collected, as the road to the deep treasures shortens and dropped chips stack at the end. Totally try to grab one of those, maybe even calculate which stack has the best average value.

- Drain the oxygen tank: if you're close to the submarine and have players behind you with a lot of treasure, it could be a good idea to pick up even Type 1 treasure chips just to drain the tank faster and make them drown. Of course, this could always backfire and get you drowned instead.

- Empty the oxygen tank: this strategy is quite different from the one above and a bit extreme: if you're too far to make it back or someone with a bit too much treasure is trying to reach the submarine, or even if you are already leading with treasures from the anterior rounds, you can try to drown everyone by grabbing as many treasures as you can. You might not manage to drown everyone, but it's worth it even if you pressure other players into dropping some treasure or picking up less chips!

- Turn back when you pick your first treasure: usually you'd like to keep all oxygen and full roll for the way back. You wouldn't usually want to burden yourself before even reaching the desired depth. A drawback of this strategy is that on the way back there's a bigger probability that you will land on an empty space and won't be able to pick up any treasure, so it might be better to do it when there are less people playing or when others are using the same strategy and leaving the return path full.

- This is almost the opposite of a tactic, but it is worth mentioning for the heuristic agent: based on manual gameplay sessions and discussions with other players, it was observed that the order of play (the starting sequence of players) appears to have negligible impact on the final win probability. Unlike Chess or Connect-4, where there is either a clear first-move advantage or different tactic for the first versus the second player, the stochastic nature of the dice and the shared

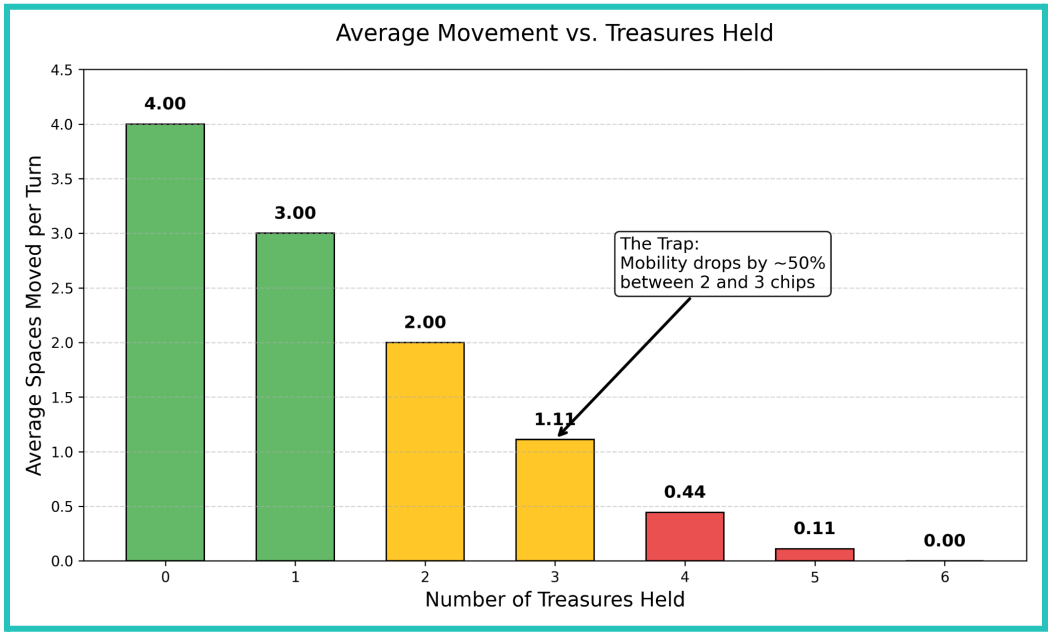
oxygen pool balances the initiative. The state space at t=0 is assumed to be neutral, requiring no artificial handicaps for model training.

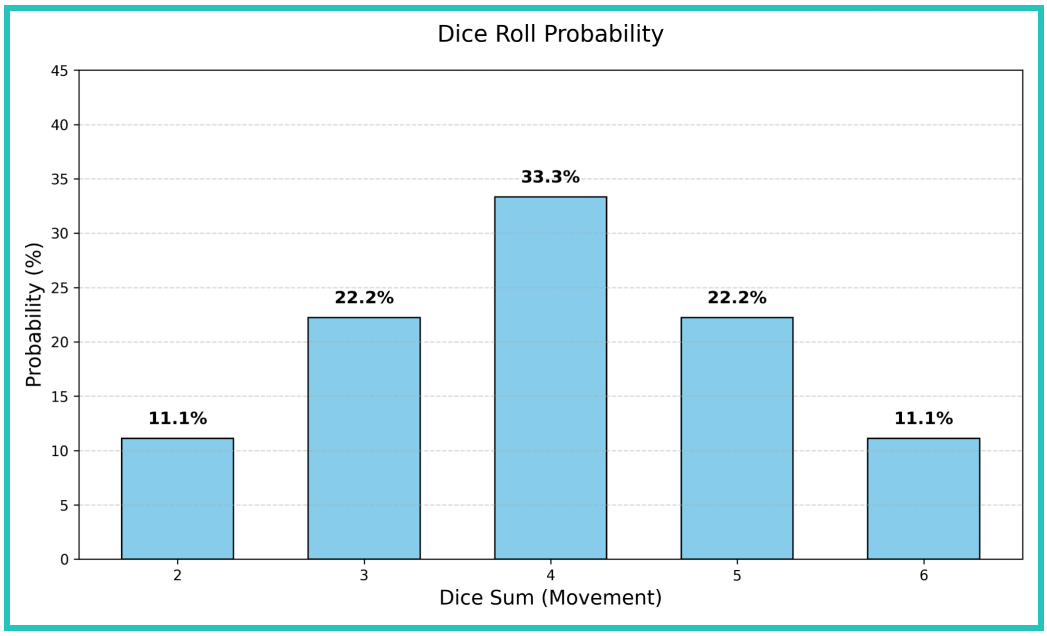
- Don’t be greedy: this one might be obvious, but having 3 or more chips might make it impossible for you to get back to the submarine; 4 chips taken will mean on average you will not move at all. With 6 chips you will never move.

Since carrying treasure chips b subtracts directly from movement speed (v = roll - b) and the expected roll of E[move] = 4, a mathematical "Greed Limit" can be found:

- Safe Load (b≤2)
  Players maintain positive expected movement.

- Critical Load (b≥3)
  Expected movement drops to 1 or less. Carrying three treasures results in a 33% chance of stalling entirely.





- This is an edge case, but at a tie the player with the most high-level treasures wins. If this is not settled then the game ends in a draw.

Approach

[references:
https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
https://en.wikipedia.org/wiki/AlphaGo_Zero
https://en.wikipedia.org/wiki/AlphaZero]

Heuristics Agent

The heuristics agent implements only a couple of the tactics enumerated above, the ones with the most impact found in manual play sessions, in different phases of the game:

- Diving Phase (going down):

    - Rule 1: Avoids collecting treasure while descending to maximize depth exploration
    - Rule 2: Exception that collects treasure early if oxygen is critically low (< 23), meaning that the other players started collecting, or if the agent is past halfway down with oxygen below 25

- Return Phase (coming back up):

    - Rule 3: Immediately returns to surface after collecting the first treasure

    - Rule 4: Collects at most one additional treasure on the way back if oxygen reserves are sufficient to survive the journey (oxygen > position)

    - Rule 5: Drops treasures if survival is uncertain (holding multiple treasures with insufficient oxygen to return)

- Movement decisions:

    - When not returning yet, prioritizes continuing descent if no treasures are held

    - Falls back to returning if at the bottom or no other moves available

The strategy is quite simple but can be used generally and it prioritizes survival and efficiency. It balances treasure collection with oxygen management, avoids greed by limiting treasure hauls, and uses a conservative approach of collecting one treasure and returning rather than maximizing depth exploration for multiple treasures.

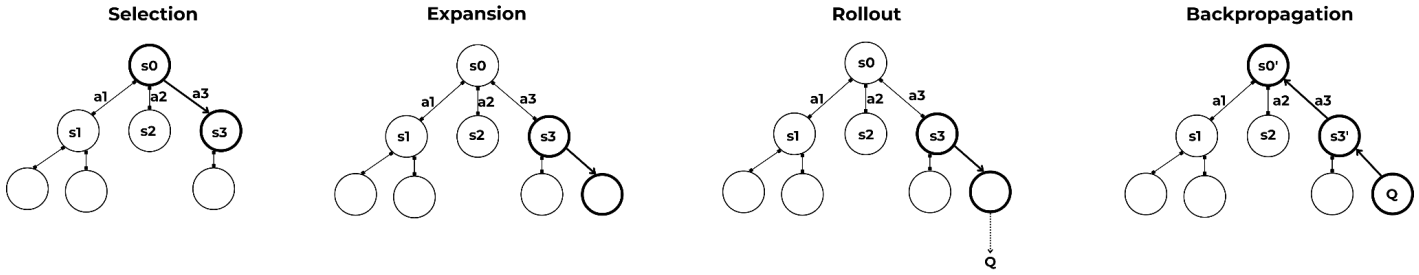
General Monte Carlo Tree Search

Monte Carlo Tree Search, or MCTS, is an heuristic search algorithm implemented with the purpose of decision making, returning the best next move given: a certain current/starting state, a way of quantifying the value of a state, and the number of possible actions taken from any state, which must be finite. It is widely used for artificial agents that play board games, most notably combined with neural networks in two-player strategy board games such as Chess, Shogi and Go. Google's AlphaGo Zero is a well-known system that uses MCTS, a Residual Neural Network and Reinforcement Learning to reach superhuman performance in the game Go, and being

outperformed later by AlphaZero in just 4 hours of training time, which was generalized for Go, Chess and Shoji.

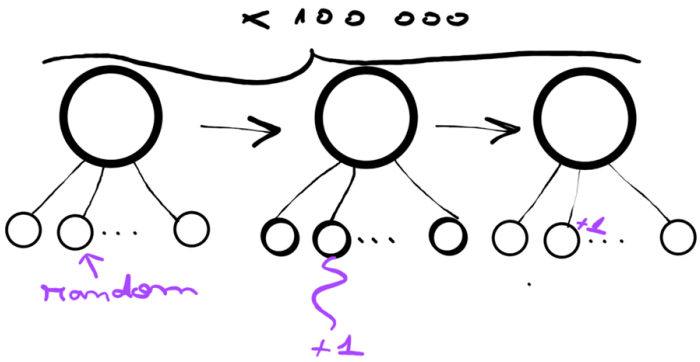
The search tree starts with a root node which contains the initial state and, on the first layer, the secondary states that can result from taking all the legal moves from the root. One of these states will end up being chosen as the best next move at the end of the algorithm. Children states are expanded with legal moves from parent nodes and explored iteratively, expanding the tree as little as possible. MCTS uses random rollouts to give value to the new expanded node, meaning that it simulates the plays of the game with random moves until an end state or a maximum depth is reached and backpropagates the value back to the expanded node. The nodes created during simulation are temporary and only the final value gets backpropagated and added to the existing parent nodes, all the way up to the root. If enough random rollouts are run, the average value of each child of the root will be a good indicator of its real value, and so the one with the highest score will be chosen.

Below is an illustration which represents a step of the learning cycle, split into the four specific sections:



Pure Monte Carlo Tree Search

One of the algorithms implemented for the Deep Sea Adventure game is the pure Monte Carlo Tree Search algorithm. This is a simplified version of the Monte Carlo Tree Search above: it uses a rollout-based approach like explained above to choose which branch has the biggest win rate and should be therefore chosen, but unlike traditional MCTS, this is "pure" in that it doesn't maintain a search tree or track game state history. The only thing present in the tree are the candidate moves coming right from the root, from which the one with the biggest win rate for the current player will be selected.



Since Deep Sea Adventure is quite a short game, the rollout simulates the play with random moves in full, until one of the players is selected winner. The reward is therefore binary, 1 if the simulated player wins and 0 otherwise, and so the win rate is calculated, per candidate move, as:

Total Wins / Rollouts

This approach is computationally simple but less sophisticated than tree-building MCTS algorithms like UCT, which typically converge faster with better move ordering.

Single Threaded Monte Carlo Tree Search

The main agent used for this project is is a modified MCTS for better suiting the Deep Sea Adventure game, but it still implements the four core MCTS phases from above:

- Selection: Traverses the tree using the Upper Confidence Bound formula to balance exploration vs. exploitation. The UCB1 formula is the following:

  $$UCB1 = wi/ni + C\sqrt{(\ln(N)/ni)}$$

  Where:

  - wi = wins/rewards accumulated by child i
  - ni = visit count of child i

- N = visit count of parent node
- C = exploration constant ($\sqrt{2} \approx 1.41$)

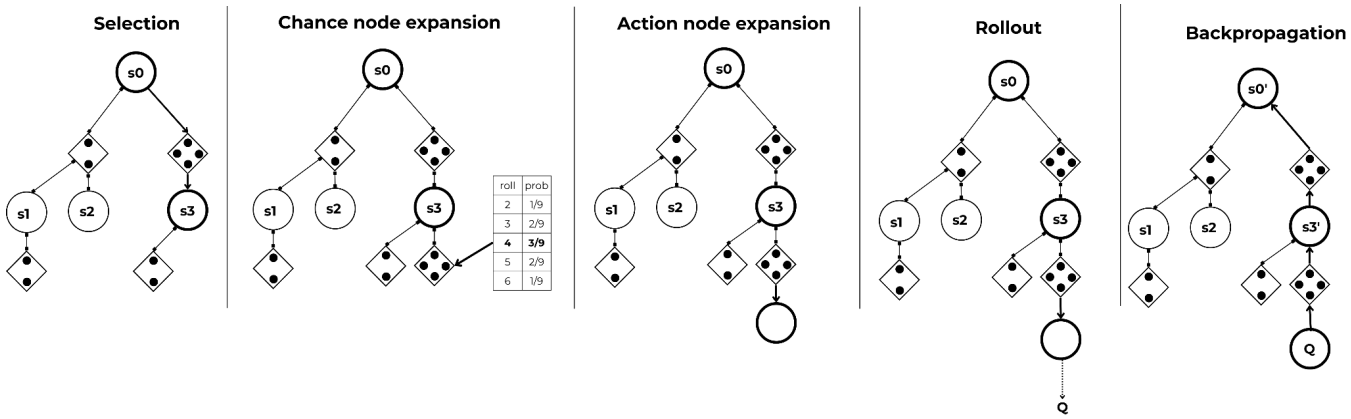
There are two components of the UCB:

1. Exploitation term: wi/ni - the average reward. Favors high-performing children.
2. Exploration term: $C\sqrt{(\ln(N)/ni)}$ - decreases as a child gets more visits. Forces trying less-visited children.

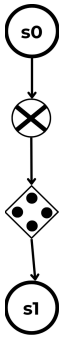
The balance prevents the algorithm from greedily picking one child (pure exploitation) while also converging toward good moves (pure exploration).

- Expansion: Randomly selects an unexpanded move and adds a child node
- Simulation: Runs a random playout to terminal state (capped at 100,000 steps)
- Backpropagation: Updates visit counts and reward sums back to the root of the tree, which is done for all players, differently from the simple MCTS above

For implementation of this game, a simple MCTS algorithm is not usable, because of the randomness of the dice rolls and the large number of players. Below is a diagram which abstractly shows the implementation of the MCTS which will contain a function that simulates the dice roll:



What's more, the action in which the player chooses when to turn back is an important one and it will represent another state of the process similar to how it is presented in this diagram:

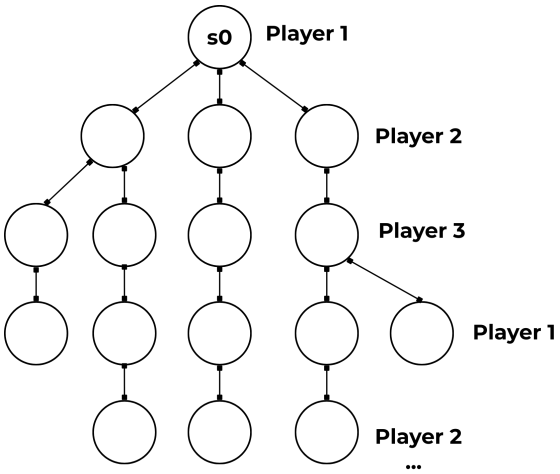


The algorithm implements multi player support, by doing the following:

- Tracks wins per player (in a vector of doubles), instead of a single score
- Each node stores rewards for all players separately

Terminal rewards: gives 1.0 to all players tied for highest points, 0.0 otherwise

The layers of the tree are also indexed accordingly, matching each layer to a player, as simplified below:



Multi Threaded Monte Carlo Tree Search

The parallelization strategy in this implementation uses thread-level parallelism with result aggregation. Root parallelisation is used, where multiple search trees are executed concurrently on separate threads. This approach minimizes synchronization overhead compared to Leaf Parallelisation. The iterations are split equally between the available threads and synchronization is done at the end to merge move statistics (totalVisits and totalWins).

Below is a diagram presenting the workload of the 32 threads increasing simultaneously for the execution of the parallelized MCTS:

For comparison, below is a diagram with the single threaded MCTS initially implemented:

Standard MCTS is **50%** faster than parallelized MCTS, which is great given that with root parallelisation, it is essentially doing **32x** Standard MCTSs.

Heuristics

In addition to the modifications above, the random rollouts of the MCTS algorithm are modified by pruning bad picks with heuristics, avoiding choices that even humans would avoid in those situations and increasing the efficiency of the algorithm.

Random Rollout MCTS is **30**% faster than Heuristic rollout MCTS, but the performance gain justifies the extra time.

Game Environment

The game environment is a custom implementation of the Deep Sea Adventure game in C++, following the original rules of the physical game.

The scope of the environment is to represent the game while minimizing memory footprint and allow the players and agent to interact with it through the functions: ‘getPossibleMoves’, and ‘doMove’.
As the name says, ‘getPossibleMoves’ returns a list of possible moves for the current state.
The function ‘doMove’ applies that move on the current state to create a new one.

The environment state contains information about the players and the current state of the board, while also keeping track of the round number, oxygen level, the last player to reach the submarine (useful when determining turn order during the next round) and the player that is about to move.

The board is represented as a dynamic contiguous container, at the start of the game it contains 32 tiles (8 of each level) and it shrinks with each round.

Each tile contains a TreasureStack, which contains all the available treasure on that tile(the treasure is just an integer that represents the level of the treasure, and its value is determined when collected).
This structure allows the environment to correctly implement the rule where a "stack" of three dropped items only counts as one item for movement and oxygen penalties.

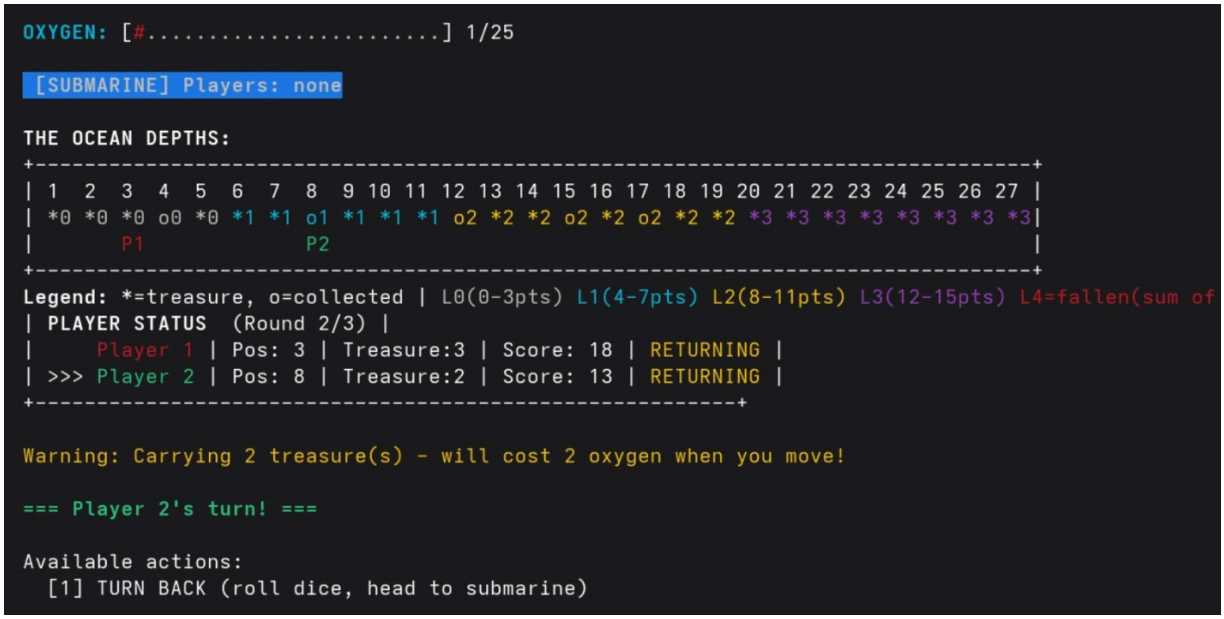
Besides the TreasureStack, the tile keeps track of its occupancy and flipped status. A tile is flipped when the treasure has been collected. A tile is occupied if a player is currently on it.
Occupied tiles are simply ‘jumped’ over during movement (actually we simply don’t count occupied tiles when calculating traveled distance during movement).

Dice rolls are handled via a thread-local random number generator (`std::mt19937`).

Each player has a position (position 0 being the submarine), a number of points, an inventory that consists of a list of TreasureStacks and 2 flags that mark whether the player is dead and whether the player is returning.

At the end of a round(when all players have reached the submarine or the oxygen level is 0), the round counter is incremented, the players' treasures are converted to points(if they reached the submarine), dead players are brought back from the dead and flown into the submarine, flipped tiles are removed, the treasure of dead players is redistributed at the bottom and the oxygen is back to 25.

The player that starts the next round is the last one to reach the submarine, if no one reached the submarine then it's the player that started first the round before.



Above is a screenshot of the game environment visualization during play. The board with the players' positions and all needed information are encompassed in the terminal.

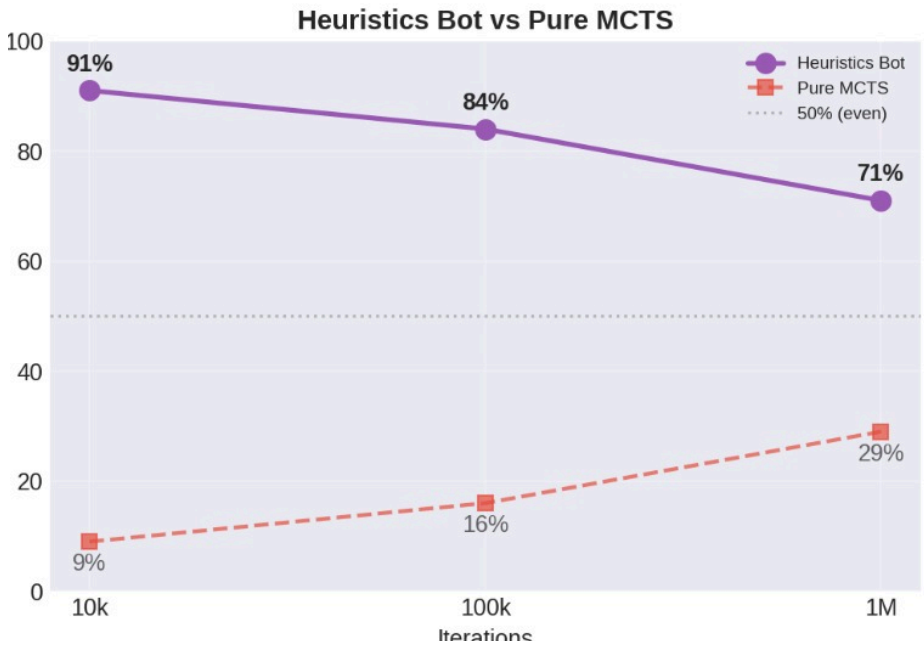
Hardware Specifications & Limitations

The system is designed for an AMD Ryzen 9 9950X (16 cores, 32 threads). The final version of the MCTS algorithm needs to be run parallelized for best time efficiency.

Evaluation & Comparison Methods

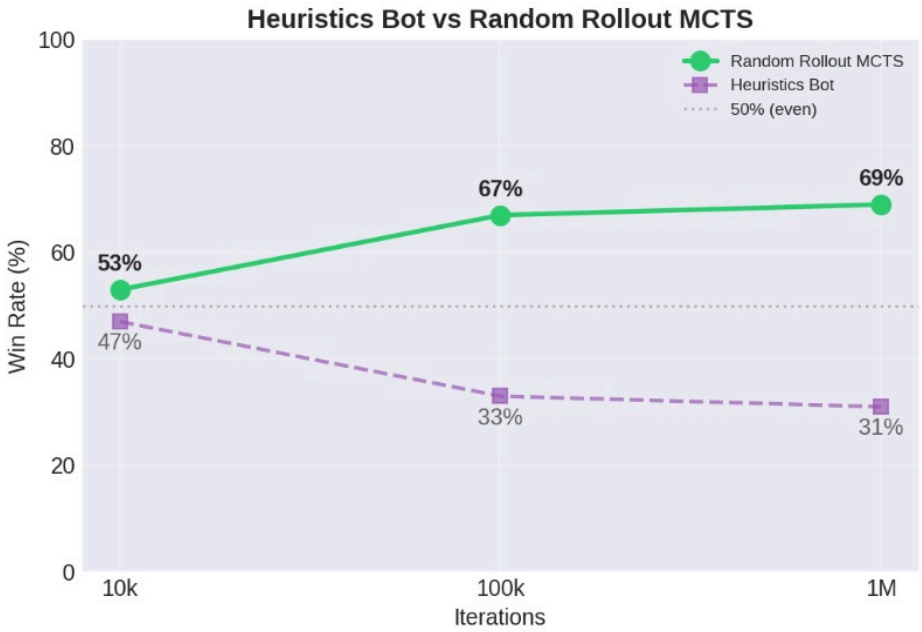
For evaluation, the different game agents were compared against each other for 100 matches, in the following order:

- Heuristics Bot vs Pure MCTS:
  The first two agents created were the Heuristics Bot and the Pure MCTS, because they were simpler and a bit less experimental to implement.



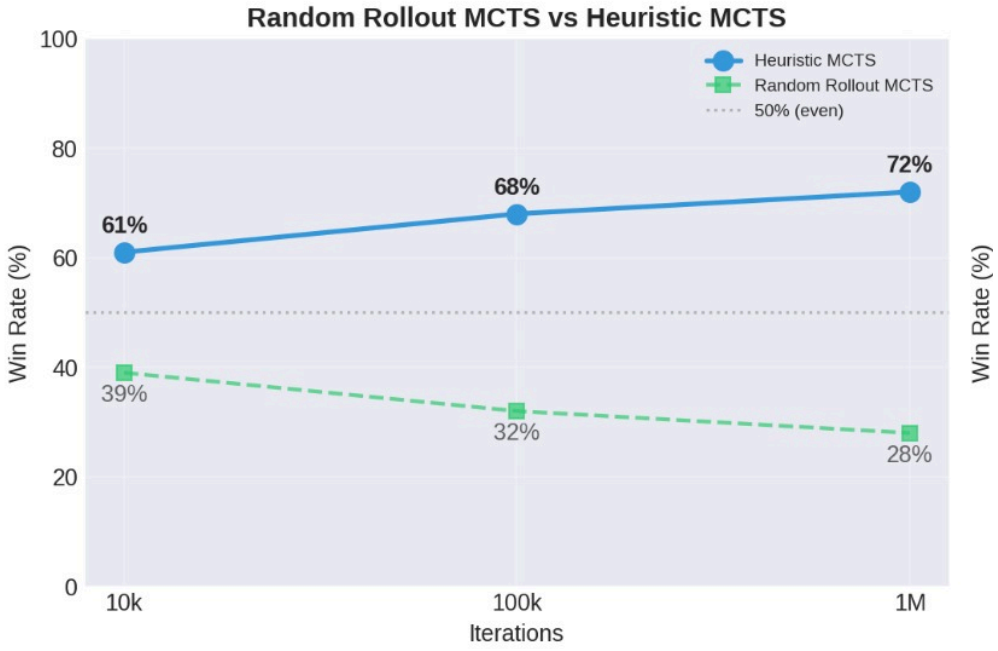
  As seen above, the Heuristics Bot has a win rate way higher than the Pure MCTS one, which is a clear tell that the game techniques implemented were valid.

- Random MCTS vs Heuristics Bot:



Even though the heuristics found were efficient, the MCTS turned out to be far more performant, especially as it progressed into its training. Therefore, this model was chosen for further modifications.

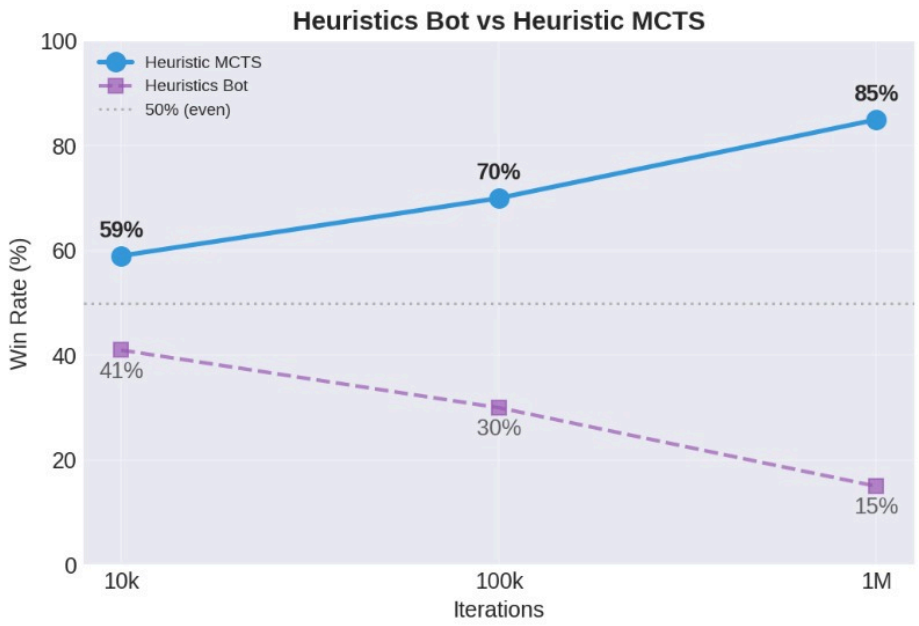
- Random MCTS vs Heuristics MCTS



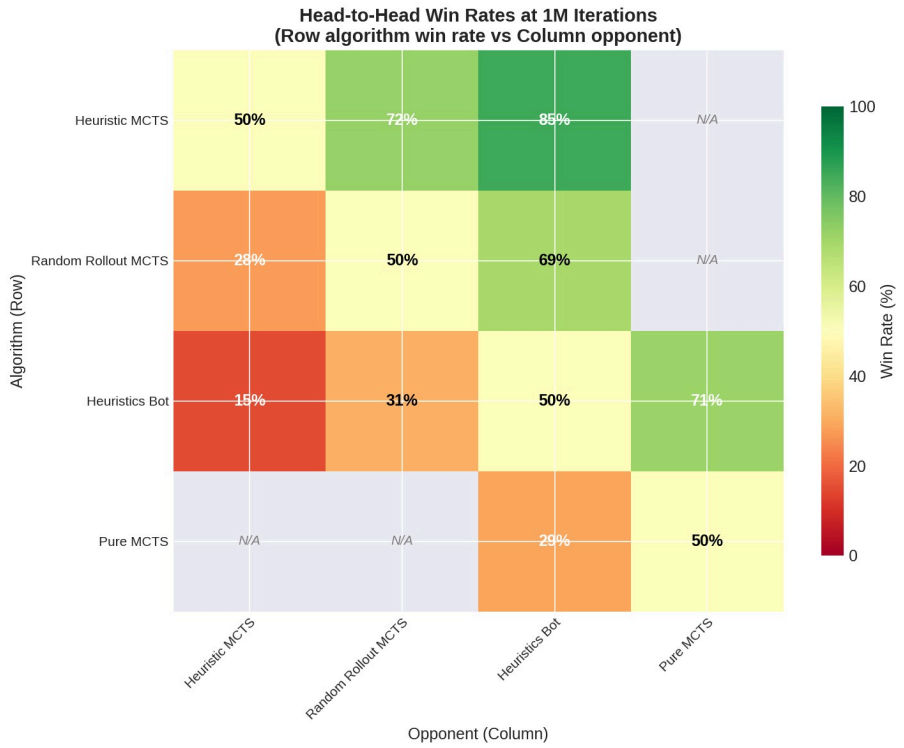
As for a last modification, it turns out adding the heuristics pruning to the MCTS model has had a great improvement over the results. This is the final model that was modified for this project, achieving a satisfactory performance when observed during play.

- Heuristics Bot vs Heuristics MCTS

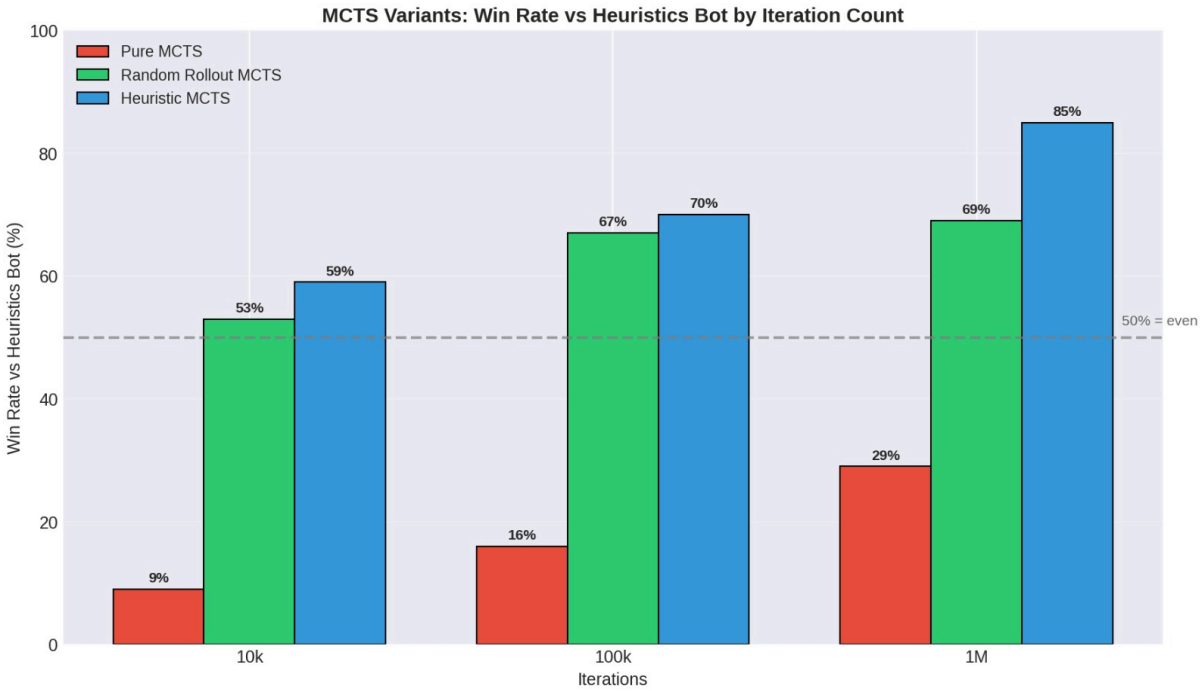
  For a last check, the Heuristics MCTS was also tested against the first Heuristics Bot, and as we can see below, the performance is indeed the best so far:



Below is a table containing all of the above for a better comparison of the algorithms. The parallelized MCTS using heuristics for pruning has the most win rates:



And below is a table comparing the three most performant algorithms implemented in the project:



Conclusions & Future Work

First of all, the project itself was very fun to do. It was interesting to see how the different algorithms play against each other and how actual results and techniques can be found even in such a random game, which is more luck-based than not. Another interesting thing was testing the techniques and heuristics I used during manual gameplay: it turns out they were really efficient!

There are a few improvements I plan to bring to this project in the future:

- Adding neural networks to the Heuristics MCTS model to (hopefully) further improve its time performance
- Since a perfect algorithm for this game is not possible or extremely hard to do, I would like to benchmark the final model by making it play matches against humans; a web deployment or making an easier way to add the AI agent to my real-life plays of the game could be helpful in that regard
- A more appealing interface would be a final, polishing step for the project