Battleship-Monte-Carlo-Simulation

- Statistische Bewertung der Schießstrategien

-Kurs: B-SDC Simtools

-Technische Hochschule Nürnberg

-Damien Rutz & Manuel Nseguet

-Wintersemester 2025/2026

-Einreichungsdatum: 09.01.2026

1. Einleitung

Battleship ist ein klassisches Spiel, das durch verborgene Informationen und inhärente Zufälligkeit gekennzeichnet ist. Aufgrund der unbekannten Platzierung der Schiffe hängt der Ausgang des Spiels stark von der gewählten Schießstrategie ab. Dies macht Battleship zu einem geeigneten Beispiel für die statistische Bewertung von Entscheidungsstrategien unter Unsicherheit.

2-Problem-Aussage

In diesem Projekt wird ein Monte-Carlo-Simulationsansatz verwendet, um verschiedene Schießstrategien im Spiel zu analysieren und zu vergleichen, Battleship.By durch die Simulation einer großen Anzahl unabhängiger Spiele wird es möglich, statistisch sinnvolle Leistungsmetriken zu erhalten. Das Hauptziel der Simulation ist es, die Effizienz jeder Strategie anhand der Anzahl der erforderlichen Schüsse zu bewerten, um alle Schiffe auf einem Standard-10*10-Raster zu sniken.

3-Methodik

In diesem Werk werden zwei verschiedene Strategien betrachtet. Die erste ist eine rein zufällige Schießstrategie, die als Basisreferenz dient. Die zweite Strategie ist eine regelbasierte künstliche Intelligenz, die einen Hunt & Target-Ansatz implementiert, der ihr Verhalten basierend auf früheren Treffern anpasst. Die Ergebnisse beider Strategien werden mit statistischen Messgrößen wie Mittelwerten und Standardabweichungen verglichen.

Dieser Bericht präsentiert die Problemformulierung, die Simulationsmethodik, die Softwareimplementierung und eine detaillierte Analyse der erhaltenen Ergebnisse.

4-Implementierung

4.1 – Zufällige Schießstrategie(game-logic.py)

Die Datei implementiert game_logic.py eine **rein zufällige Schießstrategie**, die als Basisreferenz zur Bewertung fortgeschrittenerer Ansätze dient. Diese Strategie nutzt keine Vorinformationen und behandelt jeden Schlag als eigenständiges, zufälliges Ereignis.

Zu Beginn jeder Simulation wird ein neues Spielbrett erstellt, bei dem die Schiffe zufällig gemäß der vordefinierten Flottenkonfiguration platziert sind. Die Gesamtzahl der Schiffszellen wird berechnet, und die Simulation läuft weiter, bis alle Schiffssegmente getroffen wurden.

Alle möglichen Kartenkoordinaten werden mit einem vordefinierten Zufallszahlengenerator generiert und zufällig gemischt, um die Reproduzierbarkeit sicherzustellen:

Koordinaten = [(r, c) für r in Bereich (Reihen) für c in Bereich (Querschnitte)]

Koordinaten = Liste(rng.Permutation(Koordinaten))

Während der Simulationsschleife wählt der Algorithmus sequentiell Koordinaten aus der gemischten Liste aus und erhöht den Schusszähler für jeden Versuch. Ein Treffer wird registriert, wann immer eine ausgewählte Zelle ein Schiffssegment enthält:

während hits_count < total_targets:

r, c = koordinaten.pop()

shots_fired += 1

Die Simulation endet, sobald alle Schiffszellen getroffen wurden und die Gesamtzahl der abgefeuerten Schüsse zurückgegeben wurde. Dieser Wert wird später als wichtigste Leistungskennzahl verwendet und dient als Bezugspunkt zur Bewertung der Effizienz der Smart Shooting-Strategie.

4.2 Smarter KI-Strategie-Jagd- und Zielalgorithmus

Die Datei implementiert game_logic_smart.py eine regelbasierte künstliche Intelligenz für die Schlachtschiffsimulation unter Verwendung einer **Jagd- und** Zielschießstrategie. Im Gegensatz zur zufälligen Basislinie hält dieser Ansatz eine interne Darstellung des Spielzustands bei und passt seine Entscheidungen auf Basis zuvor beobachteter Ergebnisse an.

Der Algorithmus verwendet ein Knowledge Board, um den Zustand der Zelle zu speichern und zwischen unbekannten Positionen, Fehlschlägen und erfolgreichen Treffern zu unterscheiden:

STATE_UNKNOWN = 0

STATE_MISS = 1

STATE_HIT = 2

Während der Simualtionen funktioniert die Strategie in zwei Modi. Im **Hunt-Modus** sucht der Algorithmus nach neuen Schiffen mit einem paritätsbasierten Schachbrettmuster, das den Suchraum reduziert, indem er nur abwechselnde Gitterzellen anvisiert:

wenn (r + c) % 2 == 0:

r, c = dein, uc

Sobald ein Schiffssegment getroffen wird, schaltet der Algorithmus in **den Zielmodus.**In dieser Phase werden orthogonale benachbarte Zellen einer Prioritätswarteschlange hinzugefügt und zuerst getestet, sodass der Algorithmus effizient die Ausrichtung und Größe des Schiffes bestimmen kann:

Für nr, nc in [(r + 1, c), (r - 1, c), (r, c + 1), (r, c - 1)]:

target_queue.append((nr, nc))

wenn alle Schiffssegmente erkannt sind, werden die umliegenden Zellen gemäß den Platzierungsregeln als Fehlseiten markiert, und der Algorithmus kehrt in den Jagdmodus zurück. Die Simulation läuft weiter, bis alle Schiffszellen getroffen sind und die Gesamtzahl der abgefeuerten Schüsse als primäre Leistungsmetrik für den statistischen Vergleich zurückgegeben wird.

4.3-Spiel-Aufbau-Board und Schiffsplatzierung

Die Datei game_setup.py ist dafür verantwortlich, die Battleship-Spielumgebung zu initialisieren, indem sie ein gültiges Spielbrett erstellt und Schiffe nach vordefinierten Regeln platziert. Dieses Modul umfasst alle setupbezogenen Funktionen und sorgt für eine klare Trennung zwischen Board-Initialisierung und Schießstrategien.

Das Spielbrett wird als numerische Matrix mit einer festen Größe von 10×10 dargestellt. Die Flottenkonfiguration legt sowohl die Länge als auch die Anzahl der Schiffe fest, die auf das Board gebracht werden sollen. Jeder Schiffstyp wird nach seiner Länge codiert, was eine effiziente Identifikation während der Simulation ermöglicht.

Bevor ein Schiff platziert wird, überprüft der Algorithmus, ob an der gewählten Position genügend Platz vorhanden ist. Zusätzlich zu Grenzkontrollen wird um jedes Schiff ein einzelliger Sicherheitspuffer ertvingt, um benachbarte Platzierungen gemäß den Standardregeln der Schlachtschiffe zu verhindern:

Wenn 0 <= nr < Reihen und 0 <= nc < Kols und Brett[nr, nc] != 0:

Rückkehr False

Die Schiffsplatzierung erfolgt mit einem randomisierten Prozess. Die Reihenfolge, in der die Schiffe platziert werden, ist zufällig verteilt, um den Bias zu reduzieren, und jedes Schiff erhält eine zufällige Ausrichtung und Ausgangsposition. Wenn innerhalb einer begrenzten Anzahl von Versuchen keine gültige Platzierung gefunden werden kann, wird die gesamte Initialisierung des Spielbretts neu gestartet, um Korrektheit und Robustheit zu gewährleisten:

während True:

Board = np.zeros(BOARD_SIZE, dtype=int)

Sobald alle Schiffe erfolgreich platziert sind, wird das vollständig initialisierte Board zurückgegeben. Dieses randomisierte, aber regelkonforme Setup stellt sicher, dass jeder Simulationslauf von einer einzigartigen Konfiguration beginnt und dabei durch den eingespritzten Zufallszahlengenerator reproduzierbar bleibt.

4.4-Monte Carlo Simulationsmotor

Die Datei implementiert monte_carlo.py das Monte-Carlo-Simulationsframework, das zur statistischen Bewertung von Schießstrategien verwendet wird. Anstatt ein einzelnes Schlachtschiffspiel zu analysieren, führt die Simulation eine große Anzahl unabhängiger Spiele aus und aggregiert die Ergebnisse, um sinnvolle Leistungsmesswerte wie Mittelwert, Varianz und Standardabweichung zu erhalten.

Um sowohl Zufälligkeit als auch Reprodubilität über parallele Prozesse hinweg zu gewährleisten, verlässt sich die Implementierung auf NumPys SeedSequence. Ein Masterseed wird verwendet, um unabhängige Kind-Seeds zu erzeugen, jeweils einen für jede Simulationsausführung. Jeder Arbeitsprozess erstellt dann seinen eigenen Zufallszahlengenerator aus dem zugewiesenen Seed:

master_seq = SeedSequence(base_seed)

child_sequences = master_seq.spawn(n_simulations)

Für die Leistung werden Simulationen parallel mit **ProcesspoolExecutor ausgeführt** und die Arbeitslast auf die verfügbaren CPU-Kerne verteilt.

Jeder Arbeiter führt die ausgewählte Simulationsfunktion (Zufallsstrategie oder intelligente KI) aus und gibt die Anzahl der Kurzschlüsse zurück, die zum Versenken aller Schiffe benötigt werden:

mit ProcessPoolExecutor(max_workers=n_cores) als Executor:

results_list = list(executor.map(_worker_task, tasks, chunksize=chunk))

Nach Abschluss aller Durchläufe werden die resultierenden Schusszählungen in ein NumPy-Array umgewandelt und mit grundlegenden statistischen Indikatoren zusammengefasst. Diese Werte

repräsentieren das endgültige Output, das zum Vergleich beider Strategien verwendet wird:

Return {"avg": np.mean(results), "variance": np.var(results), "std_dev": np.std(results)}

Dieses Design bietet eine effiziente und reproduzierbare Möglichkeit, verschiedene Schießstrategien unter unterschiedlichen zufälligen Schiffsplatzierungen zu vergleichen, was für eine robuste quantitative Analyse unerlässlich ist.

4.5 – Hauptprogramm und Experimentpipeline

Die Datei *dient main.py* als zentraler Einstiegspunkt des Projekts und orchestriert den gesamten Experiment-Workflow. Es koordiniert die Durchführung von Monte-Carlo-Simulationen, übernimmt die Ergebnisaggregation und erzeugt sowohl numerische als auch visuelle Leistungsvergleiche zwischen den implementierten Schießstrategien.

Zu Beginn des Programms werden globale Simulationsparameter definiert, darunter die Anzahl der Simulationen und ein fester Basis-Seed. Diese Parameter gewährleisten Reproduzierbarkeit und konsistente experimentelle Bedingungen:

N_SIMULATIONS = 100000

BASE_SEED = 42

Nach Ausführung der Monte-Carlo-Simulationen sowohl für die Zufallsstrategie als auch für die intelligente KI werden die Ergebnisse an Visualisierungs- und Berichtsfunktionen weitergegeben. Ein vergleichendes Histogramm wird erstellt, um die Verteilung der für die Strategie benötigten Aufnahmen zu veranschaulichen, wobei die Mittelwerte direkt in der Abbildung hervorgehoben werden:

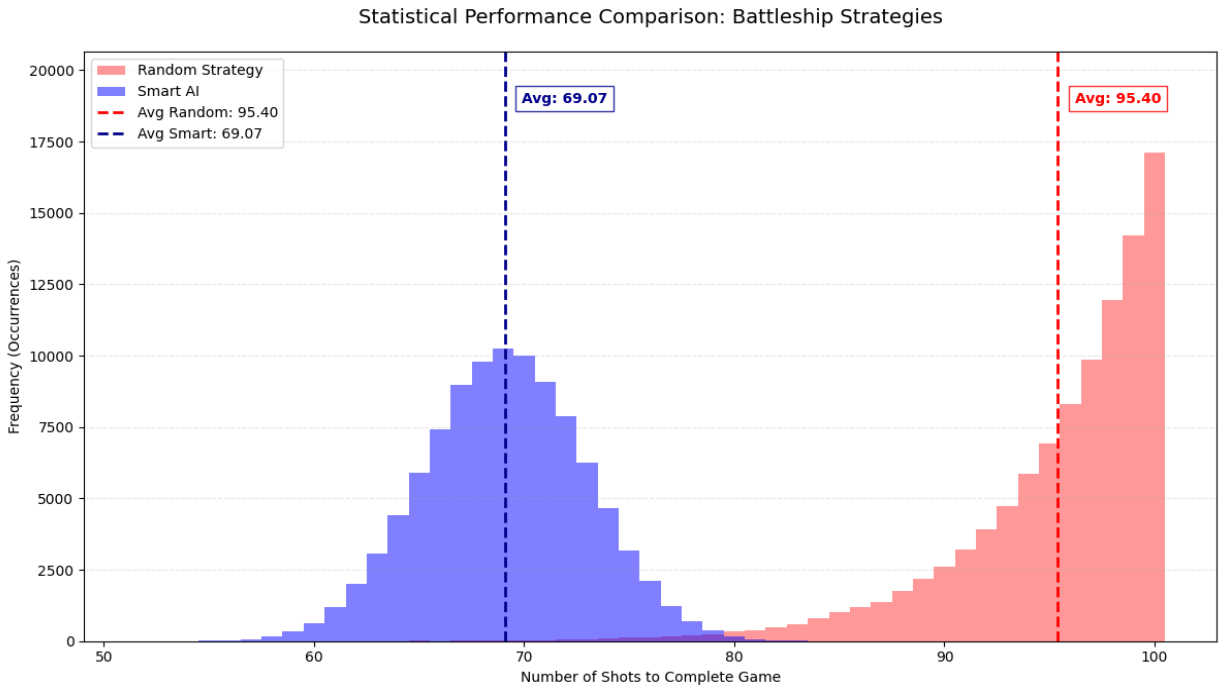
save_histogram(res_random["raw_data"], res_smart["raw_data"])

Zusätzlich zur visuellen Ausgabe druckt das Programm eine formatierte statistische Vergleichstabelle auf die Konsole. Diese Tabelle fasst wichtige Leistungsindikatoren wie durchschnittliche Schüsse, Varianz und Standardabweichung für beide Strategien zusammen und ermöglicht einen prägnanten quantitativen Vergleich:

print_comparison_table(res_random, res_smart)

Die main()-Funktion integriert alle Komponenten in eine einzige Experimentpipeline. Durch die Trennung von Simulationslogik, Visualisierung und Ausgabeformatierung in dedizierte Funktionen bleibt das Programm klar, modular und leicht zu erweitern.

5-Ergebnisse



Die obige Tabelle zeigt die Verteilung der Anzahl der Schüsse, die benötigt werden, um alle Schiffe für beide Schussstrategien zu versenken. Die intelligente KI benötigt im Durchschnitt durchweg weniger Schüsse als die zufällige Grundlinie, was auf eine deutliche Effizienzsteigerung hindeutet.