
# Bachelorarbeit

# 2048 lösen: MCTS und Expectimax mit domänenspezifischen Heuristiken im Vergleich

Mohammed Zain Maqsood

29. November 2024

Erstgutachter: Prof. Dr. Benjamin Blankertz
Zweitgutachter: Dr.-Ing. Stefan Fricke

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Neurotechnologie

# Kurzfassung

2048 ist ein populäres stochastisches Puzzle, das aufgrund der einfachen Spielregeln und kombinatorischen Komplexität Gegenstand aktueller Forschung ist. In dieser Arbeit werden zwei Verfahren sowie optimierte Versionen zur Lösung des Spiels vorgestellt. Diese sind zum einen Expectimax und zum anderen Monte Carlo Tree Search. In einer optimierten Variante wird Expectimax durch eine Transpositionstabelle zur Vermeidung redundanter Zustände erweitert. Zudem wird MCTS im MCTSE-Algorithmus um Zufallsknoten ergänzt, um die Stochastizität des Spiels besser zu modellieren. Darüber hinaus werden beide Verfahren mit domänenspezifischen Heuristiken kombiniert, um deren Leistung weiter zu optimieren. Nach der Implementierung werden einerseits die klassischen Varianten mit den optimierten Varianten verglichen, andererseits werden die optimierten Varianten untereinander gegenübergestellt. Für einen fairen Vergleich wird MCTSE ein Zeit-Budget zugewiesen, innerhalb dessen ein Zug berechnet werden muss. Die Ergebnisse deuten darauf hin, dass beide Verfahren in der Lage sind, das Spiel zu lösen. Insbesondere gelingt es Expectimax in Kombination mit einer Transpositionstabelle für ein geringes Tiefenlimit, das Spiel mit einer höheren Wahrscheinlichkeit zu lösen als MCTSE mit Zeit-Limit. Allerdings kann MCTSE bei einem höheren Zeit-Budget bessere Resultate erzielen und erzeugt zudem weniger Knoten im Suchbaum.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1 Einleitung

## 1.1 Spielbeschreibung

Das Spiel 2048 [4] ist ein stochastisches Puzzle, das standardmäßig auf einem 4x4 Brett gespielt wird. Auf diesem befinden sich Kacheln, die Zweierpotenzen als Werte haben. Zu Beginn des Spiels befinden sich zwei zufällige Kacheln auf dem Spielbrett. Diese können den Wert 2 oder 4 annehmen. Dabei liegt die Wahrscheinlichkeit, dass eine Kachel mit dem Wert 2 erscheint, bei 0,9 und 4 bei 0,1. Anschließend werden diese auf zufällig ausgewählte freie Felder gesetzt [9].

Der Spieler kann nach links, rechts, oben oder unten wischen. Dies sind die einzigen Aktionen, die während eines Zugs möglich sind. Mit diesen werden alle Kacheln so weit wie möglich in die entsprechende Zugrichtung verschoben. Kollidieren dabei zwei Kacheln mit demselben Wert, die sich am nächsten am Rand der Zugrichtung befinden, verschmelzen diese zu einer neuen Kachel, die als Wert die Summe der beiden kollidierten Kacheln hat. Zudem gilt für eine neue Kachel, die während eines Zugs aus zwei anderen Kacheln entstanden ist, dass diese nicht mit einer dritten Kachel gleichen Werts im selben Zug verschmolzen werden kann. Kacheln, die in Zugrichtung blockiert sind, bleiben an ihrem Ort. Dabei blockieren sich Kacheln, die nicht den gleichen Wert aufweisen. Bewegt sich mindestens eine Kachel in eine Richtung, handelt es sich um einen validen Zug. Nach jedem validen Zug erscheint eine neue zufällige Kachel auf einem freien Feld nach dem Prinzip wie anfangs beschrieben [9].

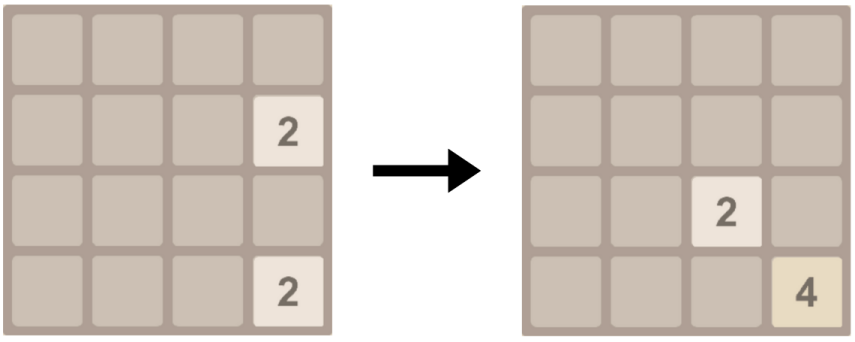


Abbildung 1.1: Zug nach unten: Beide 2-Kacheln verschmelzen zu einer neuen 4-Kachel und neue 2-Kachel erscheint

Existiert kein valider Zug, hat der Spieler verloren. Wie der Titel des Spiels bereits andeutet, ist das Ziel, die 2048-Kachel zu erreichen. Dann hat der Spieler gewonnen, das Spiel läuft aber weiter, da durchaus Kacheln mit höheren Werten erreichbar sind. Darüber hinaus existiert ein Punktestand, der sich beim Verschmelzen von zwei Kacheln um deren Summe erhöht [9].

## 1.2 Relevanz und bisherige Literatur

Das Puzzle ist aufgrund der simplen Spielregeln und zugleich kombinatorischen Komplexität Gegenstand aktueller Forschung, die es zum Ziel hat, Agenten zu entwickeln, die Kacheln mit möglichst hohen Werten erreichen. Aufgrund der stochastischen Natur des Spiels verwenden viele dieser Agenten Algorithmen, wie Expectimax oder Monte Carlo Tree Search (MCTS) als Basis. Die besten unter ihnen verwenden fortgeschrittene Methoden aus dem Bereich des Reinforcement Learning wie das Temporal Difference Learning (TD-Learning) [5][6][12] oder optimierte Expectimax-Ansätze in Kombination mit Heuristiken [13]. Der aktuell beste Agent ist in der Lage, die 32768-Kachel mit einer Erfolgsrate von 72% und einer durchschnittlichen Punktzahl von 625377 zu erreichen [5]. Darüber hinaus findet das Spiel auch Anklang in der Informatiklehre und wird beispielsweise verwendet, um Konzepte aus dem Bereich Algorithmen- und Datenstrukturen zu vermitteln [9].

## 1.3 Zielsetzung

Ziel der Arbeit ist es, das Spiel mit höchster Wahrscheinlichkeit algorithmisch zu lösen, das heißt, der Agent soll in der Lage sein, die 2048-Kachel mit konsistentem Erfolg zu erreichen. Um dies zu realisieren, wird einerseits ein optimierter Expectimax-Ansatz verwendet und zum anderen kommt MCTS(E) zum Einsatz. Beide Ansätze werden mit domänenspezifischen Heuristiken kombiniert, um die Suche zielgerichteter zu gestalten. Ferner soll festgestellt werden, welcher der beiden optimierten Verfahren besser zum Lösen des Spiels geeignet ist.

## 1.4 Struktur der Arbeit

Nachdem bereits das Spielprinzip erklärt wurde, werden im nächsten Kapitel die domänenspezifischen Heuristiken, Algorithmen und die Transpositionstabelle vorgestellt. Daraufhin werden die genannten Methoden in C++ implementiert und die Simulationsergebnisse systematisch anhand diverser Kriterien bewertet und verglichen. Anschließend werden die Ergebnisse diskutiert, weitere Optimierungsansätze besprochen und letztlich ein Fazit gezogen.

# 2 Methoden

## 2.1 Domänenspezifisches Wissen

In diesem Abschnitt werden drei Heuristiken vorgestellt, die bewerten, wie vorteilhaft ein Brettzustand ist, indem diesen ein numerischer Wert zugeordnet wird. Diese finden beispielsweise Anwendung in dem Agenten von Xiao [13], der mit seinem Agenten sehr solide Resultate erzielen konnte. Die Heuristiken sind Teil der Evaluationsfunktion und finden später im Expectimax- und (zum Teil) im MCTS(E)-Algorithmus Anwendung. Die Wahl dieser ist ausschlaggebend für den Erfolg beider Algorithmen.

### 2.1.1 Heuristiken

#### Monotonicity

Diese Heuristik bewertet, wie geordnet das Spielbrett ist. Sind die Kachelwerte auf- bzw. absteigend entlang der Zeilen bzw. Spalten, handelt es sich um einen monotonen Brettzustand. Dabei ist es irrelevant, welche Orientierung die Monotonie folgt. Diese Strategie neigt dazu, Kacheln mit höheren Werten in einer Ecke zu sammeln und insgesamt das Brett geordneter zu gestalten. Dies ist genau die Kachelanordnung, die es zu erreichen gilt, denn Spielbretter, in denen Kacheln keiner strikten Ordnung folgen, neigen häufiger dazu, dass das Spiel frühzeitig endet, da es aufgrund der Unordnung schwieriger wird, Kacheln zu verschmelzen. Ein Beispiel für ein monotones Spielbrett ist in Abb. 2.1 dargestellt.

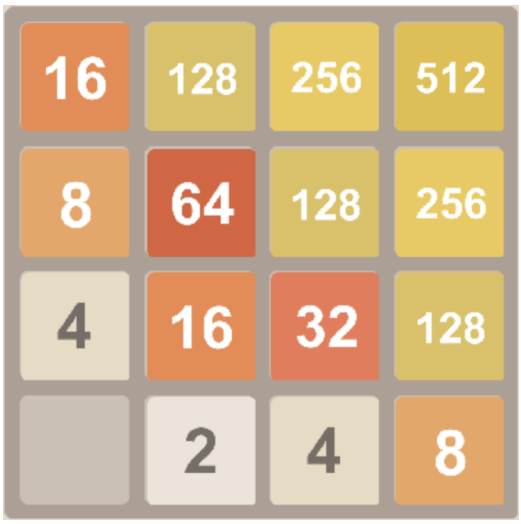


Abbildung 2.1: Spielbrett mit hoher Monotonie

Die Werte sind von links nach rechts aufsteigend bzw. von oben nach unten absteigend angeordnet. Eine mögliche Implementierung könnte so aussehen, dass man für alle vier Orientierungen paarweise benachbarte Kacheln auf diese Ordnung prüft und dabei zählt, welche Paare diese erfüllen. Anschließend wird die Orientierung mit dem höchsten Wert ausgewählt [7]. In Algorithm 1 wurde der Pseudocode für eine mögliche Implementierung der Monotonicity-Heuristik dargestellt. Diese berechnet die Anzahl monotoner Kacheln für alle Zeilen und Spalten des Spielbretts. Haben zwei benachbarte Kacheln einen unterschiedlichen Wert, wird die entsprechende Variable inkrementiert. Falls diese den gleichen Wert aufweisen, werden beide Variablen für die entsprechenden Richtungen inkrementiert (Else-Block), damit keine Richtung gegenüber der anderen bevorzugt wird. Zum Schluss werden alle Richtungen entsprechend der vier Variablen zu den vier Orientierungen kombiniert und das Maximum ausgewählt.

**Algorithm 1** Monotonicity Heuristik

```
1:  function ComputeMonotonicty(state)
2:      left_to_right ← 0
3:      right_to_left ← 0
4:      top_to_bottom ← 0
5:      bottom_to_top ← 0
6:
7:      for i = 1 to m do                          ▷ Iteriere über die Zeilen
8:          for j = 1 to n do                      ▷ Iteriere über die Spalten
9:              if j < n − 1 then
10:                 if state[i][j] > state[i][j + 1] then
11:                     left_to_right ← left_to_right + 1
12:                 else if state[i][j] < state[i][j + 1] then
13:                     right_to_left ← right_to_left + 1
14:                 else
15:                     left_to_right ← left_to_right + 1
16:                     right_to_left ← right_to_left + 1
17:             if i < m − 1 then
18:                 if state[i][j] > state[i + 1][j] then
19:                     top_to_bottom ← top_to_bottom + 1
20:                 else if state[i][j] < state[i + 1][j] then
21:                     bottom_to_top ← bottom_to_top + 1
22:                 else
23:                     top_to_bottom ← top_to_bottom + 1
24:                     bottom_to_top ← bottom_to_top + 1
25:
26:     all_orientations ← [left_to_right + bottom_to_top, right_to_left +
    bottom_to_top, right_to_left + top_to_bottom, left_to_right + top_to_bottom]
27:     monotonicity ← max(all_orientations)
28:     return monotonicity
```

### Smoothness

Die Monotonicity-Heuristik erzeugt zwar Spielbretter, die insgesamt geordneter sind, aber um Kacheln mit großen Werten zu erreichen, müssen davor viele verschmolzen werden. Damit sich Kacheln verschmelzen können, müssen diese benachbart sein und den gleichen Wert haben. Genau solche Spielbretter versucht die Smoothness-Heuristik zu priorisieren, indem beispielsweise versucht wird, die Differenz von benachbarten Kacheln so klein wie möglich zu halten. Entspricht diese Differenz dem Wert 0, folgt daraus, dass es sich um zwei Kacheln mit identischem Wert handelt.

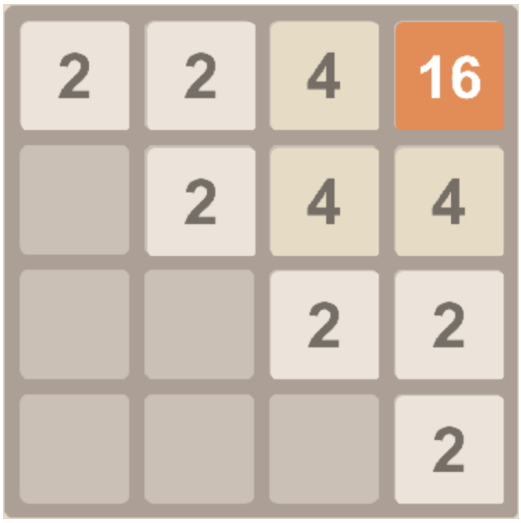


Abbildung 2.2: Spielbrett mit gutem Smoothness-Wert

Eine mögliche Implementierung könnte so aussehen, dass für zwei benachbarte Kacheln die Differenz und danach der Betrag gebildet wird. Hat ein Spielbrett einen hohen negativen Smoothness-Wert, folgt daraus, dass es benachbarte Kacheln mit größerem Werteunterschied gibt. Ist dieser nahe dem Wert 0, weist das Spielbrett einen guten Smoothness-Wert auf.

**Algorithm 2** Smoothness Heuristik

```
1: function ComputeSmoothness(state)
2:     smoothness ← 0
3:     for i = 1 to m do                          ▷ Iteriere über die Zeilen
4:         for j = 1 to n do                      ▷ Iteriere über die Spalten
5:             if j < n − 1 then
6:                 smoothness ← smoothness − abs(state[i][j]) − state[i][j + 1])
7:             if i < m − 1 then
8:                 smoothness ← smoothness − abs(state[i][j]) − state[i + 1][j])
9:     return smoothness
```

### Empty-Cells

Wie der Name bereits andeutet, ermittelt diese Heuristik die Anzahl leerer Felder, die sich auf dem Brett befinden. Intuitiv versucht ein menschlicher Spieler, diesen Wert zu maximieren, denn je mehr

leere Felder existieren, desto unwahrscheinlicher ist es, dass kein valider Zug mehr möglich ist. Aufgrund ihrer Einfachheit ist diese Heuristik leicht zu implementieren und effizient zu berechnen.

### 2.1.2 Evaluationsfunktion

Alle drei eingeführten Heuristiken sind Strategien, die intuitiv angewendet werden [9][10][13], um Kacheln mit hohen Werten zu erreichen. Um einen Brettzustand adäquat zu bewerten, bietet es sich daher an, diese in einer Evaluationsfunktion zu kombinieren. Darüber hinaus ist es sinnvoll, jede Heuristik zu gewichten. Beispielsweise bringt ein streng monotones Brett recht wenig, wenn nicht genug Kacheln existieren, die verschmolzen werden können. Oder der Agent verfolgt einen Greedy-Ansatz und versucht, so viele leere Felder wie möglich zu erzeugen, aber es herrscht keine Monotonie auf dem Brett. Daher ist es wichtig festzulegen, wie viel Einfluss jede Heuristik ausübt. Eine Evaluationsfunktion, die alle drei Heuristiken mit den entsprechenden Gewichten kombiniert, ist in Algorithm 3 dargestellt.

**Algorithm 3** Evaluationsfunktion

```
1: function EvaluateState(state)
2:     v1 ← x · ComputeMonotonicity(state)
3:     v2 ← y · ComputeSmoothness(state)
4:     v3 ← z · ComputeEmptyCells(state)
5:     result ← v1 + v2 + v3
6:     return result
```

Die Parameter $x$, $y$ und $z$ gewichten jeden Heuristikwert entsprechend. Dadurch kann der Einfluss jeder Heuristik auf das Spielbrett kontrolliert werden. Ferner können alle drei Heuristikwerte in einer einzigen verschachtelten For-Schleife, die über die Zeilen und Spalten iteriert, berechnet werden. Für die Lesbarkeit des Pseudocodes wurden diese getrennt präsentiert.

## 2.2 Expectimax-Suche

### 2.2.1 Algorithmus

Expectimax ist ein rekursiver, tiefenlimitierter Suchalgorithmus [10], der wie der Minimax-Algorithmus einen Suchbaum schrittweise aufbaut und diesen nach dem besten Zug durchsucht. Der Expectimax-Baum besitzt als Wurzelknoten einen Max-Knoten, der den maximierenden Spieler repräsentiert und jenen Kindknoten mit dem höchsten Wert wählt. Im Unterschied zu Minimax besitzt ein Max-Knoten keine Min-Knoten als Kinder, sondern sogenannte Zufallsknoten. Diese modellieren ein Zufallsereignis, wie beispielsweise einen Würfelwurf. Im Gegensatz zum Max-Knoten berechnet dieser die Summe des Produkts aus dem Ergebnis der Rekursion eines möglichen Folgezustands und der Wahrscheinlichkeit für dessen Auftreten. Jedem Zufallsknoten wird also der Erwartungswert der möglichen Folgezustände zugewiesen.
Terminalzustände beschreiben einen Spielzustand, in dem kein valider Zug möglich ist. Diese werden durch Blattknoten repräsentiert, also Knoten, die sich auf der untersten Ebene des Suchbaums befinden und die keine Kinder besitzen. Den Blattknoten wird durch eine Evaluationsfunktion ein

numerischer Wert zugeordnet, der den Spielzustand bewertet [11]. Dieser wird in dem rekursiv aufgebauten Spielbaum nach oben gereicht. Darüber hinaus kann auch je nach Spielkomplexität oder Präferenz festgelegt werden, dass Blattknoten durch eine maximale Rekursionstiefe erzeugt werden. Diese müssen nicht zwangsläufig Terminalzustände sein. Das kann vor allem dann sinnvoll sein, wenn der Zustandsraum sehr groß ist oder Speicher- und Rechenressourcen knapp sind. Nachdem der Algorithmus den Spielbaum aufgebaut und untersucht hat, wählt der Wurzelknoten jenen Zufallsknoten mit höchstem Knotenwert [11].

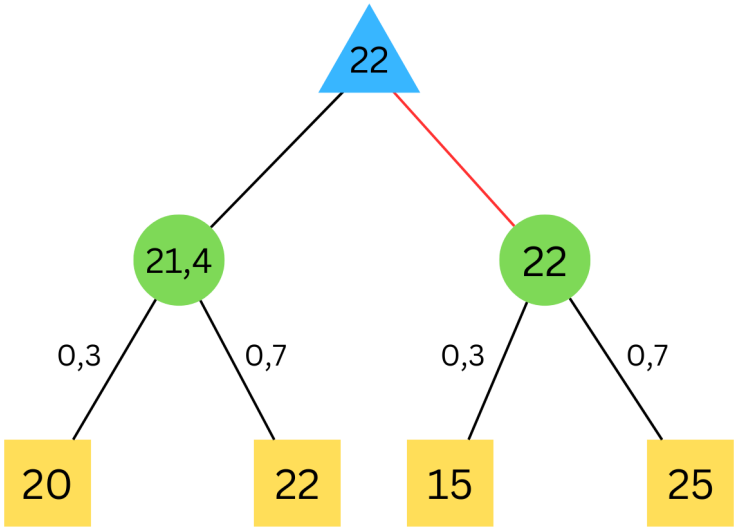


Abbildung 2.3: Beispiel für einen Expectimax-Baum

Ein Beispiel für einen Expectimax-Baum mit Max- und Zufallsknoten ist in Abb. 2.3 illustriert. Hierbei stellen das Dreieck den Maxknoten, die Kreise die Zufallsknoten und die Quadrate die Blätter dar. Die Werte in den Blattknoten sind das Ergebnis einer Evaluationsfunktion. Ausgehend vom Zufallsknoten wird für den linken Blattknoten eine Wahrscheinlichkeit von 0,3 und für den rechten von 0,7 für dessen Auftreten angenommen. Anschließend wird für beide Zufallsknoten der Erwartungswert berechnet. Da der Max-Knoten das eigene Ergebnis maximiert [11], entscheidet dieser sich für die Aktion, die durch den rechten Zufallsknoten repräsentiert wird.

### 2.2.2 Anpassung auf 2048

Nachdem der Expectimax-Algorithmus eingeführt wurde, wird dieser für unsere Zwecke auf das 2048-Puzzle angepasst. Betrachtet man am Anfang des Algorithmus einen beliebigen nicht-terminalen Zustand, repräsentiert dieser den Wurzelknoten bzw. Max-Knoten. Von diesem ausgehend sind maximal vier Aktionen in die bekannten Richtungen möglich. Die Zustände, die nach Anwendung der möglichen Aktionen auf den Zustand eines Max-Knotens resultieren, werden durch Zufallsknoten repräsentiert. Diese enthalten noch keine neue zufällige Kachel. Sei $k$ die Anzahl der freien Felder in einem Brettzustand eines Zufallsknotens. Somit ergeben sich ausgehend von diesem Zustand genau

$2k$ Folgezustände, die durch Einfügen einer neuen Kachel entstehen und durch Max-Knoten repräsentiert werden. Also $k$ Zustände, die jeweils in jedem freien Feld die 2-Kachel enthalten und analog $k$ Zustände, die die 4-Kachel enthalten. Wie bereits in der Einleitung erwähnt, beträgt die Wahrscheinlichkeit 0,9, dass eine 2-Kachel auf einem leeren Feld erscheint und 0,1 für die 4-Kachel [9]. Jeder Knotenwert eines Max-Knotens wird mit der entsprechenden Wahrscheinlichkeit der eingefügten Kachel multipliziert. Diese Produkte werden für jeden Zufallsknoten aggregiert und anschließend wird das Ergebnis durch $k$ dividiert. Nach diesem Prinzip verfährt der Algorithmus rekursiv weiter, bis ein Terminalzustand oder ein vorher festgelegtes Tiefenlimit erreicht ist. Falls der Blattknoten ein Terminalzustand ist, wird ein Strafwert zurückgegeben. Der 2048-Agent von Xiao vergibt beispielsweise einen Strafwert von 200.000 [13]. In dieser Arbeit wird der Strafwert als konstanter Wert gewählt und beträgt -10.000. Dieser wurde verhältnismäßig klein festgelegt, damit Teilbäume, die im Allgemeinen gute Zustände erzeugen, nicht komplett benachteiligt werden gegenüber anderen, die zwar keine Terminalzustände, jedoch ungünstige Zustände enthalten. Ist der Blattknoten kein Terminalzustand und das Tiefenlimit wurde erreicht, wird der entsprechende Zustand mit der Evaluationsfunktion ausgewertet. Zum Schluss wählt der Wurzelknoten die Aktion des Kindknotens mit höchstem Erwartungswert [11].

**Algorithm 4** *Expectimax-Suche* für das 2048-Puzzle

```
1: function ExpectimaxSearch(board, depth)
2:     legal_actions, new_states ← GetLegalActions(board)        ▷ Enthält ausgehende
   Zustände und entsprechende Aktionen die zu diesen führen
3:     if Length(legal_actions) = 1 then
4:         return legal_actions[1]
5:     best_utility ← −∞
6:     best_move_idx ← 0
7:     for action, new_state ∈ legal_actions, new_states do
8:         new_utility ← ChanceNode(new_state, depth)
9:         if new_utility > best_utility then
10:            best_action ← action
11:            best_utility ← new_utility
12:    return best_move
13:
14: function ChanceNode(state, depth)
15:    if depth ≤ 0 then
16:        return EvaluateState(state)
17:    total_utility ← 0
18:    empty_cells ← GetEmptyCells(state)
19:    for empty_cell ∈ empty_cells do
20:        new_state ← state
21:        InsertTile(new_state, empty_cell, 2)
22:        total_utility ← total_utility + 0,9 · MaxNode(new_state, depth − 1)
23:        InsertTile(new_state, empty_cell, 4)
24:        total_utility ← total_utility + 0,1 · MaxNode(new_state, depth − 1)
25:        RemoveTile(new_state, empty_cell)
26:    return total_utility / Length(empty_cells)
27:
28: function MaxNode(state, depth)
29:    if IsGameEnd(state) then
30:        return −10000
31:    if depth ≤ 0 then
32:        return EvaluateState(state)
33:    best_score ← −∞
34:    legal_actions, new_states ← GetLegalActions(board)
35:    for new_state ∈ new_states do
36:        new_score ← ChanceNode(new_state, depth − 1)
37:        best_score ← Max(new_score, best_score)
38:    return best_score
```

Die Funktionsweise der Expectimax-Suche ist in Algorithm 4 dargestellt. Eingabe der EXPECTIMAXSEARCH-Funktion ist der aktuelle Brettzustand des Spiels und die maximale Rekursionstiefe. Der Rückgabewert ist der Zug mit dem höchsten Erwartungswert. Da die Wahrscheinlichkeit für das Erscheinen einer 4-Kachel sehr gering ist, könnte man diese Folgezustände in der Funktion CHANCENODE vernachlässigen und damit die Laufzeit des Algorithmus deutlich verbessern. Der Vollständigkeit halber wird dies aber nicht getan. Da sich Brettzustände bei der Suche häufig doppeln, kann stattdessen eine Optimierung durch eine Transpositionstabelle durchgeführt werden.

## 2.3 Transpositionstabelle

Während der Suche nach einem optimalen Zug in einem Spielbaum kann es passieren, dass bereits untersuchte Brettzustände erneut auftreten, diese aber durch eine andere Zugsequenz erreicht werden. Solche Brettzustände, die durch mehr als eine Zugsequenz erreicht werden können, werden Transpositionen genannt [1]. Damit der Agent Transpositionen erkennt, wird jeder Brettzustand gehasht, also in eine (meist) eindeutige Zahl überführt und anschließend in einer sogenannten Transpositionstabelle (TT) gespeichert. Tritt nun dieser Brettzustand erneut auf, erzielt der gehashte Wert dieses Brettzustands einen Treffer bei einem Lookup, da dieser bereits in der TT gespeichert ist.

### 2.3.1 Zobrist-Hashing

Ein weit verbreiteter Ansatz, um ein Spielbrett effizient zu hashen, ist das Zobrist-Hashing. Zu Beginn wird eine Zobrist-Tabelle mit Pseudo-Zufallszahlen generiert. Diese enthält für jeden möglichen Kachelwert $2^1, 2^2, ..., 2^n$ (wobei $n \in \mathbb{N}^+$ typischerweise die maximale Kachel ist, die für die entsprechende Dimension des Spielbretts möglich ist) an jeder Position auf dem Spielbrett eine meist 64-Bit Pseudo-Zufallszahl. Um für ein Spielbrett nun einen Hashwert zu erzeugen, wird für jede Kachel im aktuellen Brettzustand überprüft, welche Pseudo-Zufallszahl für diese an der entsprechenden Position in der Zobrist-Tabelle gespeichert ist. Diese werden anschließend mit einer XOR-Operation verknüpft [2]. Davor wird jede Kachel mit der Vorschrift $\log 2(\text{Kachelwert}) - 1$ auf eine natürliche Zahl abgebildet, um Indexierung ohne Lücken zu gewährleisten. Aus theoretischer Sicht ist es möglich, dass Kollisionen auftreten, das heißt, zwei verschiedene Zustände weisen denselben Hashwert auf [2]. Aufgrund der Nutzung von 64-Bit-Zufallszahlen ist die Wahrscheinlichkeit jedoch extrem gering, dass diese auftreten und das Collision-Handling verschlechtert überdies die Geschwindigkeit des Algorithmus. Daher wird in dieser Arbeit auf das Collision-Handling verzichtet.

### 2.3.2 Lookup und Speicherung

Wird nun ein Zustand im Expectimax-Baum untersucht, erfolgt die Berechnung des Hashwertes nach Zobrist. Danach wird durch ein Lookup überprüft, ob der gleiche Hashwert bzw. Zustand bereits in der Transpositionstabelle gespeichert ist. Ist dies der Fall, wird anschließend überprüft, ob die gespeicherte Tiefe mindestens so hoch wie die des aktuellen Zustands ist. Falls dem so ist, kann der entsprechende Eintrag frühzeitig zurückgegeben werden, denn um diesen Spielzustand zu erreichen, waren mindestens genau so viele Züge nötig und damit ist dieser Pfad im Baum mindestens ebenso detailliert. Andernfalls wird regulär der Wert des aktuellen Knotens berechnet und die aktuelle

Tiefe in Kombination mit relevanten Informationen wie dem Knotenwert in die TT gespeichert. Die Integrierung der TT in Expectimax erfolgt über die CHANCENODE-Funktion und ist im Folgenden dargestellt.

**Algorithm 5** *Optimierte Expectimax-Variante* für das 2048-Puzzle

```
1:  function ChanceNode(state, depth)
2:      if depth ≤ 0 then
3:          return EvaluateState(state)
4:
5:      hash ← computeHash(state)
6:      entry ← TranspositionTableLookup(hash)
7:      if NOT(entry is null) then
8:          if depth ≥ entry.depth then
9:              return entry.utility
10:
11:     total_utility ← 0
12:     empty_cells ← GetEmptyCells(state)
13:     for empty_cell ∈ empty_cells do
14:         new_state ← state
15:         InsertTile(new_state, empty_cell, 2)
16:         total_utility ← total_utility + 0,9 · MaxNode(new_state, depth − 1)
17:         InsertTile(new_state, empty_cell, 4)
18:         total_utility ← total_utility + 0,1 · MaxNode(new_state, depth − 1)
19:         RemoveTile(new_state, empty_cell)
20:
21:     total_utility ← total_utility / Length(empty_cells)
22:     SaveTranspositionTableEntry(hash, total_utility, depth)
23:     return total_utility
```

## 2.4 Monte Carlo Tree Search

MCTS ist ein bewährter Ansatz, um Spiele effektiv zu lösen und findet vermehrt Einsatz in verschiedenen Domänen der Künstlichen Intelligenz (KI). Große Erfolge wurden vor allem in Kombination mit Methoden aus dem Bereich des *Deep Reinforcement Learning*, durch Computerprogramme wie AlphaGo (2016) verzeichnet. Dabei arbeitet MCTS auf kombinatorisch komplexen Zustandsräumen, die durch Suchbäume repräsentiert werden und verwendet die Monte-Carlo-Methode, bei der wiederholt zufällige Simulationen durchgeführt werden [15].

### 2.4.1 Klassisches MCTS

Im Allgemeinen ist MCTS ein iterativer Algorithmus, der einen Suchbaum aufbaut und durchsucht, bis ein vorher festgelegtes Rechenlimit wie eine Zeit-, Speicher- oder Iterationsbeschränkung erreicht

ist und die Aktion ausgehend vom Wurzel-Knoten mit dem besten Wert ausgewählt wird [3]. Dabei gibt es vier Phasen:

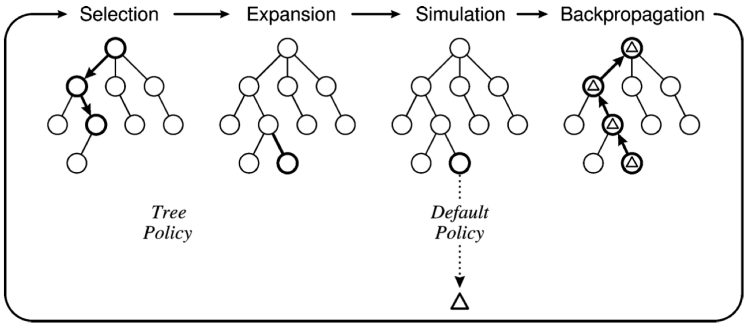


Abbildung 2.4: MCTS Phasen [3]

**Selection** - Startend vom Wurzel-Knoten wird in jeder Ebene des Suchbaums der nächste Knoten nach einer sogenannten *Tree-Policy* ausgewählt, bis ein Blattknoten erreicht ist, der nach dieser eine Expandierung oder Simulation am meisten nötig hat. Die Tree-Policy ist in Algorithm 6 durch GETBESTCHILD-Funktion dargestellt. Ein Blattknoten ist expandierbar, falls dieser einem Nicht-Terminalzustand entspricht und bereits einmal simuliert wurde [3]. Wurde dieser noch nicht simuliert, wird direkt zum Simulationsschritt gesprungen.

**Algorithm 6** MCTS - Selektionphase

```
1: function SELECT(V)
2:     while NOT(V is terminal) do
3:         if NOT(V fully expanded) then
4:             return EXPAND(V)
5:         else
6:             V ← GETBESTCHILD(V)                ▷ UCT wählt besten Knoten
7:     return V
```

**Expansion** - In diesem Schritt wird dem ausgewählten Knoten aus dem Selektionsschritt ein neuer Kindknoten hinzugefügt. Dieser entspricht einem Zustand, der durch Anwendung einer möglichen Aktion auf den Zustand des Blattknotens generiert wird. Anschließend wird dieser zurückgegeben und die Simulationsphase wird eingeleitet [15]. Für die Anpassung der EXPANDFUNKTION-Funktion auf das 2048-Puzzle wird nach dem Erzeugen eines neuen Knotens eine 2- oder 4-Kachel in den korrespondierenden Brettzustand eingefügt. Dies wurde in Algorithm 7 nicht aufgeführt, da hier der allgemeine Ansatz präsentiert wird.

**Algorithm 7** MCTS - Expansionsphase

```
1: function EXPAND(V)
2:     choose untried action a ∈ Actions(state(V))
3:     add new child V_new to V with A(state(V_new)) = a
4:     return V_new
```

**Simulation** - Im Simulationsschritt, auch als *Rollout* bekannt, wird ausgehend vom korrespondierenden Zustand des im Selektionsschritt ausgewählten Blattknotens eine Monte Carlo Simulation durchgeführt. Die *Default-Policy* beschreibt, nach welcher Strategie die Simulation vom neuen Blattknoten bis zu einem Terminalzustand durchgeführt wird [3]. Diese entspricht der *Random-Policy* in der Standardvariante von MCTS, das heißt, es werden zufällige Aktionen angewendet, bis ein Terminalzustand oder eine festgelegte *Rollouttiefe* erreicht ist. Danach wird der resultierende Zustand beispielsweise durch eine Evaluationsfunktion bewertet. Der numerische Wert, den die Evaluationsfunktion dem Zustand zuordnet, wird auch *Payoff* genannt. Eine Rollouttiefe ist vor allem dann sinnvoll, wenn Ressourcen knapp sind oder gute Ergebnisse auch mit wenigen Simulationsiterationen erreicht werden können. Ferner würde auch hier nach dem Ausführen der Aktion in Zeile 6 von Algorithm 8 eine neue Kachel in das Spielbrett eingefügt werden. EVALUATESTATE in Zeile 7 ist nicht die Evaluationsfunktion aus Abschnitt 2.1, sondern in diesem Fall wird die Empty-Cells-Heuristik als Evaluationsfunktion verwendet. Eine nähere Ausführung dazu gibt es in Kapitel 3.

**Algorithm 8** MCTS - Simulationsphase

```
1: function ROLLOUT(V, maxRolloutDepth)
2:     state ← state(V)
3:     while NOT(V is teminal or maxRolloutDepth) do
4:         legal_actions ← GETLEGALACTIONS(state)
5:         action ← CHOOSERANDOMACTION(legal_actions)
6:         state ← APPLYACTION(state, action)
7:     return EVALUATESTATE(state)          ▷ Reward für den Endzustand wird berechnet
```

**Backpropagation** - Im letzten Schritt wird der Payoff vom aktuellen Blattknoten, entlang des Pfades bis zum Wurzelknoten propagiert und währenddessen alle Knoten auf diesem Pfad aktualisiert. Das heißt, zum einen wird der Payoff auf alle Utility-Werte der Knoten auf dem Pfad addiert und zum anderen die Anzahl der Besuche um eins erhöht [15].

**Algorithm 9** Backpropagation

```
1: function BACKPROPAGATE(V, Δ)
2:     while NOT(V is null) do
3:         N(V) ← N(V) + 1
4:         Q(V) ← Q(V) + Δ
5:         V ← PARENT(V)
```

Nachdem der MCTS-Baum aufgebaut und alle Phasen durchlaufen wurden, entscheidet sich der Agent für eine Aktion. Hierzu werden die Kinder des Wurzelknotens betrachtet. Der gängigste Ansatz ist, dass der Agent diejenige Aktion des korrespondierenden Knotens wählt, der den Ausdruck $\frac{w_K}{n_K}$ maximiert.

**Algorithm 10** MCTS-Algorithmus

```
1: function MCTS(V_root)
2:     while (within computational budget and state is not terminal) do
3:         V_leaf ← SELECT(V_root)                    ▷ Selektion und Expansion
4:         Δ ← ROLLOUT(V_leaf)                        ▷ Simulation
5:         BACKPROPAGATE(V_leaf, Δ)
6:     return BESTACTION(V_root)
```

$w_K$ ist der Utility-Wert von Kind $K$ und $n_K$ ist die Anzahl der Besuche. Darüber hinaus gibt es auch andere Ansätze, eine Aktion auszuwählen, wie den *Robust Child*, bei dem der Knoten mit der höchsten Anzahl an Aufrufen genommen wird, oder der *Max-Robust Child*, der eine Kombination aus den zwei vorherigen Ansätzen darstellt [8].

### 2.4.2 Upper Confidence Bounds for Trees (UCT)

Einen maßgeblichen Anteil am Erfolg von MCTS trägt die Wahl einer angemessenen Tree-Policy, die im Selektionsschritt entscheidet, welche Knoten untersucht werden sollen. Das Ziel einer guten Tree-Policy sollte es sein, ein angemessenes Gleichgewicht zwischen Exploration (von Knoten, die nicht häufig besucht wurden) und Ausbeutung (von vielversprechenden Knoten). Ein weit verbreiteter Ansatz ist der *Upper Confidence Bounds applied for Trees* (UCT), der von Kocsis und Szepesvári (2006) vorgestellt wurde und durch folgende Formel formalisiert wird [15]:

$$\text{UCT} = \frac{w_i}{n_i} + C\sqrt{\frac{\ln N_i}{n_i}}.$$

- $w_i$ - Anzahl der Siege bzw. Summe der Payoffs des $i$-ten Knotens
- $n_i$ - Anzahl der Besuche des $i$-ten Knotens
- $N_i$ - Anzahl der Besuche des Eltern-Knotens von Knoten $i$
- $C$ - Kontrolliert das Verhältnis zwischen Exploration und Ausbeutung

Der linke Summand entspricht dem Ausbeutungsterm und sorgt dafür, dass Knoten, die häufig Siege erzielen, priorisiert werden. Der rechte Summand entspricht dem Explorationsterm und ermutigt dazu, Knoten zu betrachten, die wenig besucht wurden [3]. Die Explorationskonstante $C$ wird in der Regel mit $\sqrt{2}$ gewählt [15], kann aber je nach Performance des Algorithmus bei Bedarf angepasst werden.

### 2.4.3 Wichtige Eigenschaften

Monte Carlo Tree Search (MCTS) besitzt einige wichtige Charakteristika, die der Grund dafür sind, dass dieser eine beliebte Wahl für verschiedene Domänen ist.

1. Eine wichtige Eigenschaft ist, dass MCTS komplett ohne Evaluationsfunktionen auskommt und damit kein domänenspezifisches Wissen benötigt. Damit ist es für eine Vielzahl von Problemen geeignet, die durch einen Suchbaum modelliert werden können. Andererseits erreichen

Ansätze, die domänenspezifisches Wissen mit MCTS kombinieren, erstaunliche Ergebnisse in Spielen wie Go. Dies zum Preis von Performance, denn aufgrund der oftmals rechen intensiven Evaluationsfunktionen sind in der gleichen Zeit meist weniger Simulationen möglich [3].

2. Da nach jeder Iteration der Payoff vom aktuellen Blattknoten bis zum Wurzelknoten propagiert wird, ist gewährleistet, dass der Baum immer auf dem aktuellsten Stand ist. Dies ermöglicht es zu einem beliebigen Zeitpunkt den aktuell beste Aktion zurückgeben. Eine Verbesserung der Entscheidungsqualität ist häufig durch eine längere Laufzeit realisierbar [3].

3. Der UCT-Algorithmus erzeugt einen asymmetrischen Baum, da Knoten mit höheren Werten häufiger untersucht werden und der Baum an diesen Stellen meist in Tiefe geht, ohne andere Knoten komplett außer Acht zu lassen [3].

### 2.4.4 MCTSE

Es ist bekannt, dass MCTS sehr gut für deterministische perfect-information Spiele wie Go und Schach geeignet ist. Kutsch [8] hat jedoch herausgefunden, dass klassisches MCTS für 2048 nicht praktikabel ist, da der Agent einen stark eingeschränkten Baum aufbaut und nicht ausgehend von einer Aktion alle $2 \cdot k$ Möglichkeiten für das Erscheinen einer Kachel, wobei $k$ die Anzahl der leeren Felder auf dem Spielfeld ist, adäquat berücksichtigt. Um dies zu modellieren, hat Kutsch MCTS mit Expectimax (MCTSE) kombiniert und bessere Ergebnisse als mit dem klassischen Ansatz erzielt [8]. Da Expectimax neben Max-Knoten auch mit Zufallsknoten arbeitet, wird der MCTS-Suchbaum um diese Knotenart erweitert. Der MCTSE-Baum besteht nun aus alternierenden Ebenen von Expectimax- und Zufallsknoten. Der Expectimaxknoten repräsentiert den Zustand, der durch das Ausführen einer möglichen Aktion, ausgehend vom Brettzustand eines Zufallsknotens erreicht werden kann. Der Zufallsknoten dagegen repräsentiert einen neuen Brettzustand, der durch das Einfügen einer zufälligen Kachel in den Brettzustand des Expectimaxknoten erzeugt wird.
Diese Änderungen finden sich auch in der Selektions- und Expansionsphase wieder. In der Selektionsphase wird ein Zufallsknoten gesucht, der expandiert werden soll. Dazu wird im aktuellen Zufallsknoten überprüft, ob jede aktuell mögliche Aktion durch einen Expectimaxknoten repräsentiert wird [8].

**Algorithm 11** MCTSE - Selektionsphase

```
1:  function SELECT(V)
2:      while NOT(V terminal or maxTreeDepth) do
3:          if NOT(V fully expanded) then
4:              return EXPAND(V)
5:          V_best ← GETBESTCHILD(V)
6:          s' ← INSERTRANDOMTILE(V_best)
7:          if ∃ CHILD(V_best) with s' then
8:              V ← CHILD(V_best)
9:          else
10:             add new Chance child V_new with s' to V_best
11:             V ← V_new
12:     return V
```

Ist dies nicht der Fall, geht der Algorithmus zur Expandierung über. Andernfalls wählt der UCT-Algorithmus den besten Expectimaxknoten unter den Kindern des Zufallsknotens aus. Nun wird eine zufällige Kachel in das Spielbrett eingefügt und anschließend überprüft, ob unter den Kindern des Expectimaxknoten ein Zufallsknoten mit diesem Brettzustand existiert. Ist das der Fall, wird genau dieser für die nächste Iteration ausgewählt. Ansonsten wird ein neuer Zufallsknoten erstellt, der diesen Brettzustand repräsentiert. Die INSERTRANDOMTILE-Methode ist identisch zu der in Expectimax und fügt mit einer Wahrscheinlichkeit von 0,9 eine 2-Kachel und mit 0,1 eine 4-Kachel in den Brettzustand ein.

Wurde ein Zufallsknoten in der optimierten Selektionsphase gefunden, wird diesem bei der optimierten Expansion ein neuer Expectimaxknoten und Zufallsknoten hinzugefügt. Dem neuen Expectimaxknoten wird anschließend ein neuer Zufallsknoten hinzugefügt. Dieser repräsentiert einen neuen Brettzustand, der durch das Einfügen einer zufälligen Kachel in den Brettzustand des Expectimaxknotens erzeugt wird.

**Algorithm 12** MCTSE - Expansionsphase

1: **function** EXPAND($V$)
2: choose untried action $a \in \text{Actions}(state(V))$
3: add new Expectimax child $V_{exp}$ to $V$ with $\text{A}(state(V_{exp})) = a$
4: $s' \leftarrow$ INSERTRANDOMTILE($V_{exp}$)
5: add new Chance child $V_{new}$ with $s'$ to $V_{exp}$
6: **return** $V_{new}$

Die Simulations- und Backpropagation-Phase sind bei MCTSE identisch zur klassischen Variante.

# 3 Ergebnisse

## 3.1 Projektstruktur und Parameterwahl

Das Projekt wurde in C++ implementiert. Genutzt wurde ein g++ Compiler mit der höchsten Optimierungsstufe -O3. Ausgeführt wurden die Simulationen auf einem System mit ARM-Architektur und 8 GB RAM. Ferner ist der Projektordner in zwei Subordner gegliedert. Ein Subordner enthält die Basisfunktionalität wie die Spiellogik, Board-Datenstruktur, Transpositionstabelle und globale Variablen. Der andere enthält die Implementierungen der Algorithmen und domänenspezifischen Heuristiken.

Die Expectimax-Suche wird für die Tiefen 2 bis 6 simuliert. Des Weiteren verwendet MCTS(E) keine obere Ressourcengrenze, sondern eine Iterationsgrenze, die entscheidet, wie ausführlich die Suche sein soll. In dieser Arbeit wird MCTS(E) für die Iterationsgrenzen 500, 1000, 1500, 2000, und 2500 getestet. Außerdem verwendet dieser eine *rollout depth*, die festlegt, wie viele Züge in der Simulationsphase maximal durchgeführt werden. Diese wurde für alle MCTS(E)-Agenten auf 8 festgelegt. Dies aus dem Grund, da einerseits die Rechenressourcen stark begrenzt sind und zum anderen diese nach mehreren Testdurchläufen eine gute Balance zwischen Rechentiefe und Geschwindigkeit schien. Kann am Anfang des Expectimax- bzw. MCTS(E)-Algorithmus ausgehend vom Wurzel-Spielzustand lediglich eine Aktion durchgeführt werden, wird diese direkt zurückgegeben, ohne in die Suchphase überzugehen.

Die Parameter bzw. Gewichte für die Evaluationsfunktion entscheiden maßgeblich, wie erfolgreich der Expectimax-Agent ist. Für die Evaluationsfunktion, die bei Expectimax zum Einsatz kommt, wurden die Gewichte $x = 0,5$, $y = \frac{1}{log2(max\_tile)}$ und $z = 10$ gewählt. Dabei wurden alle Gewichte experimentell durch Testen verschiedener Werte in entsprechenden Intervallen bestimmt und wiesen die beste Performance auf. Für den MCTS(E) wurde zunächst auch als Evaluationsfunktion Algorithm 3 verwendet. Dieser erzielte jedoch unbefriedigende Ergebnisse. Da der alleinige Einsatz der Empty-Cells-Heuristik ohne Gewichtung deutlich besser performte, wurde diese als Evaluationsfunktion für MCTS(E) am Ende eines Rollouts verwendet. Als Explorationskonstante für den UCT-Algorithmus wurde der gängige Wert $C = \sqrt{2}$ gewählt, da nach einigen Testläufen dieser die besten Resultate erzeugte.

Beide Algorithmen werden anhand vier Kriterien verglichen. Diese sind die erreichten Kacheln, die erreichten Punktzahlen zum Ende eines Spiels, die Anzahl rekursiver Aufrufe bei Expectimax bzw. generierter Knoten bei MCTS(E) und die Laufzeit, die pro Zug benötigt wurde. Hierbei werden zunächst die Ergebnisse des Expectimax-Algorithmus, dann die des MCTS(E)-Algorithmus betrachtet und im Anschluss die optimierten Ansätze gegenübergestellt. Bei der Gegenüberstellung wird zuerst MCTSE mit den festgelegten Iterationsgrenzen mit dem optimierten Expectimax verglichen. Danach findet ein Vergleich zwischen beiden Verfahren statt, bei dem die Laufzeit pro Zug des MCTSE dem Expectimax-Algorithmus angeglichen wird, das heißt, jeder Zug hat ein festes Zeitlimit, in dem die-

ser berechnet werden muss. Diese Zeitlimits werden anhand der durchschnittlichen Laufzeiten pro Zug der Expectimax-Agenten mit den Tiefenlimits 2 bis 6 festgelegt. Diese Maßnahme wird vorgenommen, um beide Verfahren vergleichbar zu machen. Ferner ist anzumerken, dass die klassische und optimierte Expectimax-Variante mit TT lediglich anhand der Anzahl rekursiver Aufrufe und der Laufzeit für 5 Durchläufe verglichen werden. Grund dafür ist die extrem lange Laufzeit der klassischen Expectimax-Variante. Das bedeutet, dass die restlichen Ergebnisse davor sich ausschließlich auf die optimierte Variante beziehen. Darüber hinaus werden für jeden Agenten (außer Expectimax ohne TT) 50 Simulationen auf einem Standard 4x4 Brett durchgeführt, das heißt, vom Initialzustand bis zum Spielende.

Im Folgenden werden Kürzel im Format $\text{ALGORITHM}_{\text{Depth/Iterations}}$ verwendet. Beispielsweise bezeichnet $\text{EXP}_2$ den Expectimax-Agenten mit Tiefenlimit 2.

## 3.2 Expectimax

### 3.2.1 Erreichte Kacheln

In Tab. 3.1 sind die Simulationsergebnisse der Expectimax-Agenten für unterschiedliche Baumtiefen dargestellt. Jeder Agent konnte die 2048- und 4096-Kachel mindestens einmal erreichen. Der insgesamt beste Agent $\text{EXP}_6$ war in der Lage, in nahezu jedem Durchlauf die 2048-Kachel und in jedem dritten die 4096-Kachel zu erreichen. Darüber hinaus ist zu beobachten, dass die Agenten mit einer geraden Tiefe besser abschneiden als die ohne.

| Depth | Max Tile | Avg. Max Tile | 4096 [%] | 2048 [%] | 1028 [%] | 512 [%] |
|---|---|---|---|---|---|---|
| 2 | 4096 | 1935 | 14% | 64% | 94% | 100% |
| 3 | 4096 | 1536 | 2% | 50% | 92% | 100% |
| 4 | 4096 | 2079 | 14% | 76% | 98% | 100% |
| 5 | 4096 | 1761 | 8% | 58% | 96% | 100% |
| 6 | 4096 | 2682 | 32% | 98% | 100% | 100% |

Tabelle 3.1: Expectimax - Erreichte Kacheln

*Average Max Tile* bezeichnet hier den durchschnittlichen Wert aller maximalen Kacheln der Simulationen für die entsprechende Tiefe. Dieser liegt bei den besten Agenten $\text{EXP}_4$ und $\text{EXP}_6$ bei über 2000.

### 3.2.2 Punktestand

Separat für die geraden und ungeraden Tiefen betrachtet, ist ein Anstieg des durchschnittlichen Punktestands zu erkennen. Die Performance des $\text{EXP}_2$-Agenten ist besser als die von $\text{EXP}_3$ und $\text{EXP}_5$, obwohl letztere den Suchbaum gründlicher traversieren konnten, während der Berechnung eines Zugs. Des Weiteren ist in Abb. 3.1 erkennbar, dass $\text{EXP}_4$ im Durchschnitt einen leicht besseren Punktestand im Gegensatz zu $\text{EXP}_2$ erzielt. Gleiches gilt für $\text{EXP}_6$ im Vergleich zu $\text{EXP}_4$, wobei $\text{EXP}_6$ deutlich stärker ist und im Mittel 42.0801 Punkte erreichen konnte.

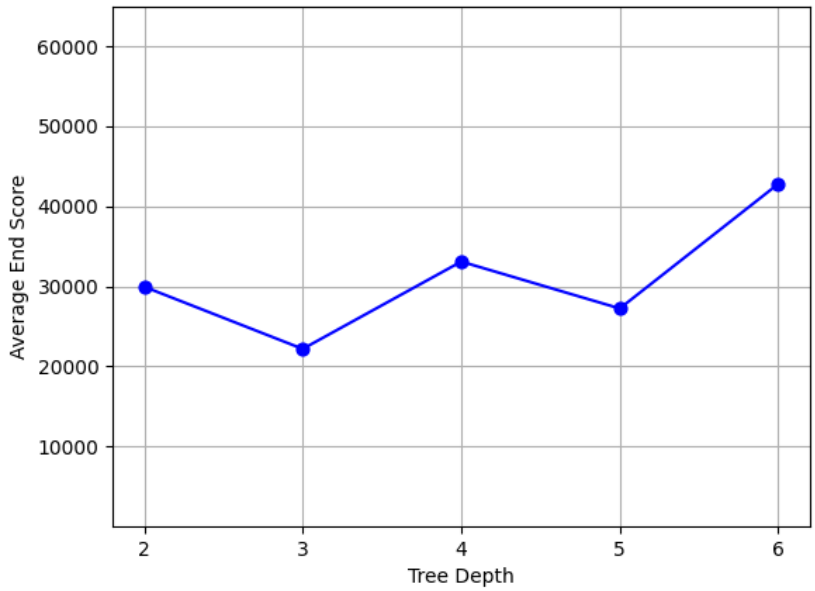


Abbildung 3.1: Expectimax - Durchschnittlicher Punktestand am Ende eines Spiels

### 3.2.3 Anzahl rekursiver Aufrufe und Laufzeit

Im Folgenden werden Resultate der Expectimax-Agenten mit Transpositionstabelle und ohne betrachtet. In der Implementierung wurde darauf verzichtet, Knoten im Suchbaum explizit zu modellieren. Stattdessen entspricht ein Knoten einem rekursiven Aufruf. Beispielsweise, wenn ausgehend vom Wurzelknoten (Max-Knoten) vier Aktionen möglich sind, wird viermal für jeden aus der Aktion resultierenden neuen Zustand die CHANCENODE-Funktion (Zufalls-Knoten) aufgerufen.
Betrachtet man die durchschnittliche Anzahl rekursiver Aufrufe pro Tiefenlimit, kann festgestellt werden, dass bei allen Agenten ohne TT deutlich mehr rekursive Aufrufe durchgeführt werden, mit Ausnahme von $EXP_2$. Dieser hat eine ähnliche Anzahl rekursiver Aufrufe im Vergleich zum Agenten ohne TT. Dies liegt daran, dass es nach zwei Zügen kaum Transpositionen gibt.

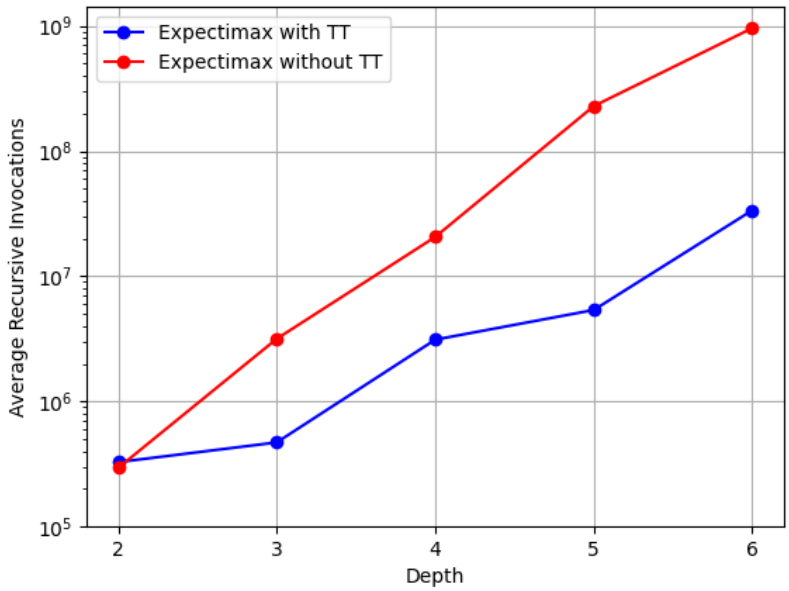


Abbildung 3.2: Expectimax mit TT vs. Expectimax ohne TT: Durchschnittliche Anzahl rekursiver Aufrufe

Bei der Tiefe 3 und 4 gibt es bereits erste Unterschiede, denn da unterscheidet sich die Anzahl der rekursiven Aufrufe um etwas weniger als eine Größenordnung. Bei der Tiefe 5 und 6 ist der Unterschied drastischer, da hier ein Unterschied von ungefähr 1,5 Größenordnungen zu erkennen ist.

Diese Ergebnisse werden auch durch die Laufzeit untermauert. $\text{EXP}_2$ mit und ohne TT benötigen durchschnittlich $0,003\frac{s}{Zug}$. Bei $\text{EXP}_3$ ist bereits eine Verbesserung von $0,02\frac{s}{Zug}$ ohne TT auf $0,005\frac{s}{Zug}$ mit TT festzustellen, also eine Beschleunigung um den Faktor 4. Eine ähnliche Beschleunigung ist bei Tiefe 4 festzustellen, denn der Agent ohne TT benötigt $0.15\frac{s}{Zug}$, wohingegen der optimierte Agent eine durchschnittliche Laufzeit von $0.035\frac{s}{Zug}$ erreicht. Aufgrund des starken Verzweigungsgrades bei Tiefe 5 und 6 steigt die Laufzeit des Agenten ohne TT ähnlich wie die Anzahl der rekursiven Aufrufe exponentiell an. Dieser benötigt für Tiefe 5 durchschnittlich $1,3\frac{s}{Zug}$ und für Tiefe 6 eine Laufzeit von $10\frac{s}{Zug}$. Die Agenten mit TT dagegen benötigen $0,058\frac{s}{Zug}$ für Tiefe 5 und $0,2\frac{s}{Zug}$ für Tiefe 6. Damit ist $\text{EXP}_6$ mit TT um den Faktor 50 beschleunigt im Vergleich zu dem entsprechenden Agenten ohne TT.

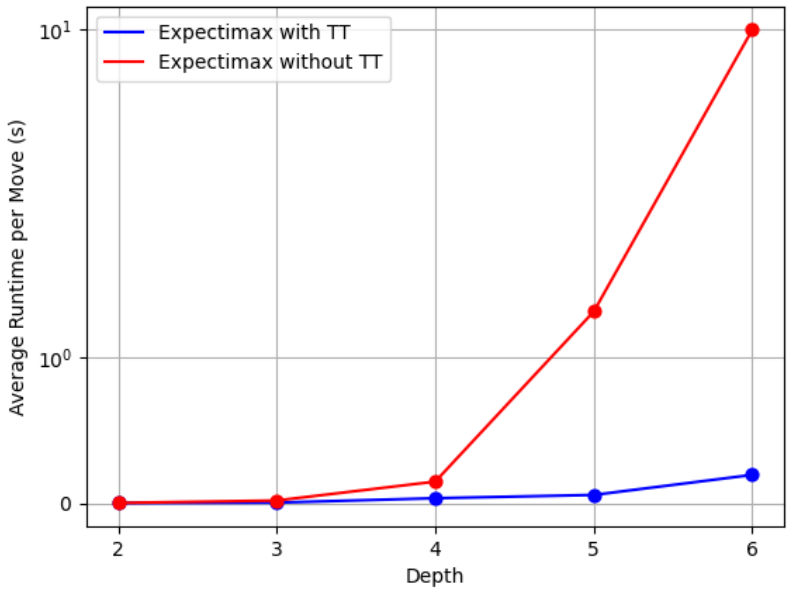


Abbildung 3.3: Expectimax mit TT vs. Expectimax ohne TT: Durchschnittliche Laufzeit pro Zug

## 3.3 MCTS vs. MCTSE

Im Folgenden wird der klassische MCTS-Algorithmus mit der optimierten Variante anhand derselben Kriterien wie in Abschnitt 3.2 verglichen.

### 3.3.1 Erreichte Kacheln

Der MCTS-Agent konnte für alle Iterationsgrenzen in beinahe jedem Durchlauf die 1024-Kachel, in mindestens jedem zweiten die 2048-Kachel und mindestens einmal die 4096-Kachel erreichen. Des Weiteren fällt auf, dass $\text{MCTS}_{2500}$ zwar etwas häufiger die 4096-Kachel erreicht, $\text{MCTS}_{1000}$ jedoch trotz deutlich weniger Iterationen vergleichbare Resultate erzielen konnte. Zudem kann $\text{MCTS}_{1000}$ häufiger die 2048-Kachel erreichen, als $\text{MCTS}_{1500}$ und $\text{MCTS}_{2000}$.

| Iterations | Max Tile | Avg. Max Tile | 4096 [%] | 2048 [%] | 1028 [%] | 512 [%] |
|---|---|---|---|---|---|---|
| 500 | 4096 | 1526 | 2% | 48% | 94% | 100% |
| 1000 | 4096 | 1792 | 2% | 72% | 98% | 100% |
| 1500 | 4096 | 1720 | 4% | 64% | 92% | 100% |
| 2000 | 4096 | 1772 | 4% | 68% | 94% | 100% |
| 2500 | 4096 | 1915 | 10% | 70% | 94% | 100% |

Tabelle 3.2: MCTS - Erreichte Kacheln

Auch alle MCTSE-Agenten weisen solide Resultate auf. Mit jedem Iterationslimit war es möglich, in den meisten Spielen die 2048-Kachel zu erzielen. $\text{MCTSE}_{2000}$ ist der insgesamt beste Agent und

konnte in mehr als der Hälfte der Durchläufe die 4096-Kachel erreichen. Des Weiteren ist auffällig, dass der schwächste MCTSE-Agent, nämlich $MCTSE_{500}$, bessere Resultate als der beste MCTS-Agent erzielt, obwohl dieser deutlich weniger Iterationen benötigt, um den nächsten Zug zu berechnen.

| Iterations | Max Tile | Avg. Max Tile | 4096 [%] | 2048 [%] | 1028 [%] | 512 [%] |
|---|---|---|---|---|---|---|
| 500 | 4096 | 2273 | 18% | 86% | 100% | 100% |
| 1000 | 4096 | 2509 | 30% | 86% | 98% | 100% |
| 1500 | 4096 | 2806 | 42% | 90% | 100% | 100% |
| 2000 | 4096 | 3113 | 54% | 96% | 100% | 100% |
| 2500 | 4096 | 2867 | 42% | 96% | 100% | 100% |

Tabelle 3.3: MCTSE - Erreichte Kacheln

### 3.3.2 Punktestand

Diese Ergebnisse spiegeln sich auch in den durchschnittlichen Punkteständen wider. Die MCTS-Agenten liegen zwischen 20.000 und 30.000 Punkten und weisen eine Verschlechterung bei $MCTS_{1500}$ und $MCTS_{2000}$ gegenüber $MCTS_{1000}$ auf, trotz mehr Iterationen pro Zug. Mit $MCTS_{2500}$ ist erneut eine Steigerung des durchschnittlichen Punktestandes festzustellen.
Die MCTSE-Agenten bis $MCTSE_{2000}$ weisen eine konstante Verbesserung des Punktestands auf. Der $MCTSE_{2500}$-Agent zeigt dagegen eine Verschlechterung zum $MCTSE_{2000}$-Agenten. $MCTSE_{2000}$ kann im Schnitt 50.000 Punkte erreichen und ist damit der insgesamt beste Agent.

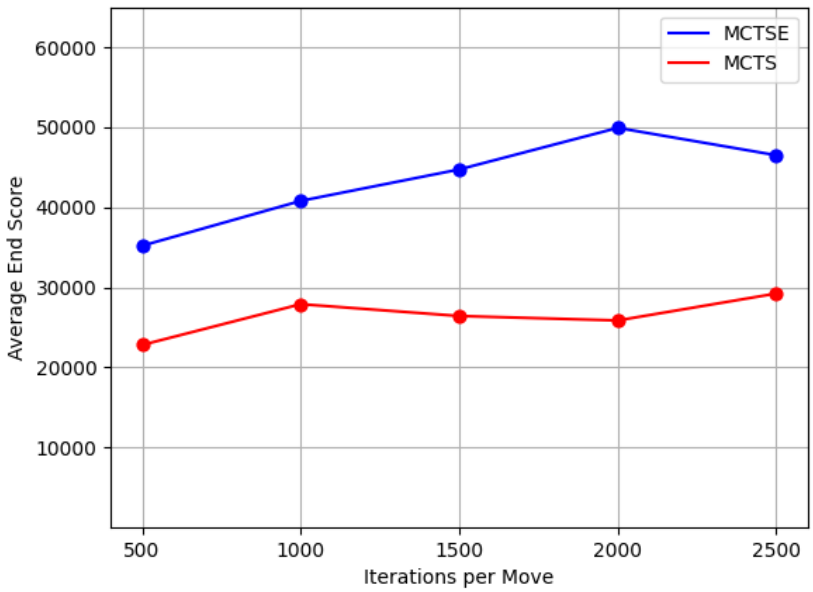


Abbildung 3.4: MCTSE vs. MCTS: Durchschnittlicher Punktestand am Ende eines Spiels

### 3.3.3 Anzahl generierter Knoten

Abb. 3.5 veranschaulicht die durchschnittliche Anzahl der generierten Knoten pro Spiel in Abhängigkeit der jeweiligen Iterationsgrenzen. Der Anstieg der generierten Knoten bleibt für beide Algorithmen unterhalb eines linearen Wachstums in Bezug auf die Anzahl der Iterationen. Konkret erzeugt $\text{MCTSE}_{500}$ ungefähr das 3,3-Fache an Knoten im Vergleich zu $\text{MCTS}_{500}$. Für die Iterationsgrenzen 1000 und 1500 ist der Unterschied erheblicher, weil $\text{MCTSE}_{1000}$ etwa das 3,9-Fache und $\text{MCTSE}_{1500}$ das 4,7-Fache an Knoten im Vergleich zur Standardvariante für die entsprechenden Iterationsgrenzen erzeugt. Abschließend generiert $\text{MCTSE}_{2000}$ und $\text{MCTSE}_{2500}$ das 4,9-Fache an Knoten im Kontrast zu $\text{MCTS}_{2000}$ und $\text{MCTS}_{2500}$.

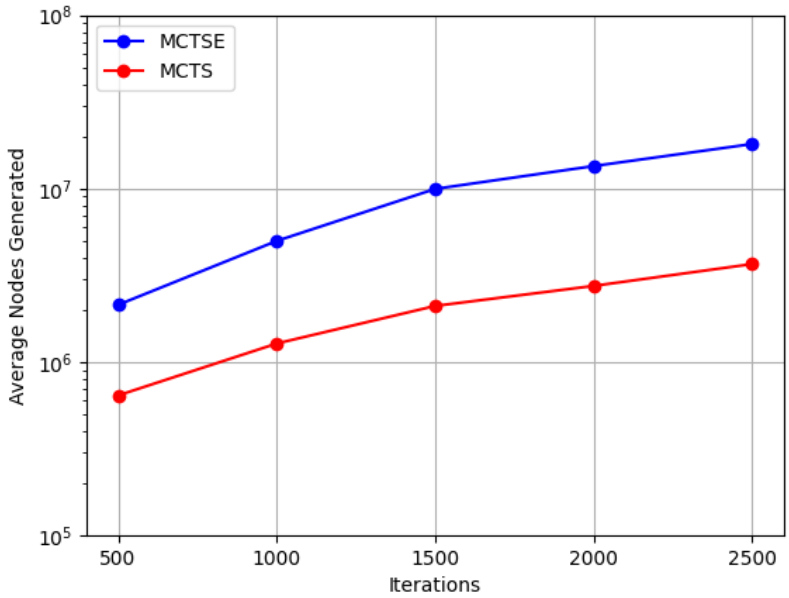


Abbildung 3.5: MCTSE vs. MCTS: Durchschnittliche Anzahl generierter Knoten pro Spiel

### 3.3.4 Laufzeitvergleich

Die Laufzeit beider Algorithmen steigt ungefähr linear zur Anzahl der Iterationen und beide Kurven folgen einem ähnlichen Trend, wie in Abb. 3.6 zu erkennen ist. Dabei benötigt $\text{MCTS}_{500}$ etwa $0,11\frac{s}{Zug}$ und $\text{MCTSE}_{500}$ $0,14\frac{s}{Zug}$. Die Agenten mit der höchsten Iterationsgrenze benötigen jeweils $0,52\frac{s}{Zug}$ für $\text{MCTS}_{2500}$ und $0,55\frac{s}{Zug}$ für $\text{MCTSE}_{2500}$ im Durchschnitt. Betrachtet man alle Laufzeiten im Mittel, berechnet der MCTS-Algorithmus insgesamt etwas schneller einen Zug.

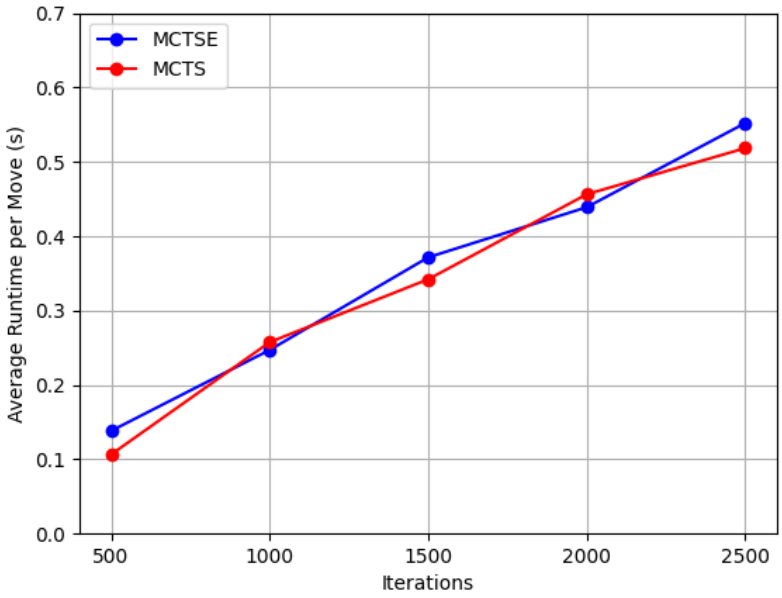


Abbildung 3.6: MCTSE vs. MCTS: Durchschnittliche Laufzeit pro Zug

## 3.4 MCTSE vs. Expectimax

### 3.4.1 Bisherige Resultate

Bevor die Ergebnisse des MCTSE mit angeglichenem Zeitbudget vorgestellt und mit Expectimax (mit TT) verglichen werden, folgt eine kurze Gegenüberstellung der bisherigen Resultate der optimierten Varianten.
Tab. 3.4 zeigt im Überblick die erreichten Kacheln der optimierten Verfahren. Hierbei können alle EXP-Agenten in mindestens jedem zweiten Spiel die 2048-Kachel und der beste unter ihnen in fast jedem Spiel erzielen. Zudem erzielt der beste Agent in ungefähr jedem dritten Durchlauf die 4096-Kachel. Darüber hinaus ist 4096 der höchste Wert der Kacheln, die erreicht wurden. Im Vergleich dazu konnten alle MCTSE-Agenten die 2048-Kachel wesentlich konsistenter erreichen. $\text{MCTSE}_{2000}$ und $\text{MCTSE}_{2500}$ können diese sogar in fast jedem Durchlauf erzielen. Darüber hinaus konnte der beste MCTSE-Agent in ungefähr jedem zweiten Spiel die 4096-Kachel erzielen. Auch bei MCTSE war 4096 der höchste erreichte Kachelwert.

| **Depth** | **Max Tile** | **Avg. Max Tile** | **4096 [%]** | **2048 [%]** | **1028 [%]** | **512 [%]** |
|---|---|---|---|---|---|---|
| 2 | 4096 | 1935 | 14% | 64% | 94% | 100% |
| 3 | 4096 | 1536 | 2% | 50% | 92% | 100% |
| 4 | 4096 | 2079 | 14% | 76% | 98% | 100% |
| 5 | 4096 | 1761 | 8% | 58% | 96% | 100% |
| 6 | 4096 | 2682 | 32% | 98% | 100% | 100% |
| **Iterations** | | | | | | |
| 500 | 4096 | 2273 | 18% | 86% | 100% | 100% |
| 1000 | 4096 | 2509 | 30% | 86% | 98% | 100% |
| 1500 | 4096 | 2806 | 42% | 90% | 100% | 100% |
| 2000 | 4096 | 3113 | 54% | 96% | 100% | 100% |
| 2500 | 4096 | 2867 | 42% | 96% | 100% | 100% |

Tabelle 3.4: MCTSE vs. Expectimax: Gemeinsame Ergebnisse der erreichten Kacheln

In Abb. 3.7 sind die restlichen Ergebnisse der optimierten Agenten in neuer Kombination dargestellt. Die linke Abbildung zeigt die durchschnittlichen Punktstände am Ende eines Spiels und untermauert die zuvor besprochenen Resultate bzgl. der erreichten Kacheln. Konkret erzielt jeder MCTSE-Agent einen höheren Punktestand, als der korrespondierende EXP-Agent. In der mittleren Abbildung fällt auf, dass der EXP-Agent für niedrige Tiefen deutlich weniger Knoten erzeugt, als der MCTSE-Agent. Dafür steigt dessen Knotenanzahl mit dem Anstieg der Tiefe rasch an. Die Knotenanzahl des MCTSE-Agenten dagegen flacht mit zunehmenden Iterationen ab. Überdies ist eine Änderung des Verhältnisses bei Tiefenlimit 6 erkennbar. Bei diesem erzeugt $\text{EXP}_6$ ungefähr 3,36 Mio. Knoten und $\text{MCTSE}_{2500}$ ungefähr 1,81 Mio. Knoten. Und abschließend ist in der rechten Abbildung zu erkennen, dass die Laufzeit von Expectimax eher exponentiell wächst, wohingegen die des MCTSE eher linear wächst und letzterer im Mittel deutlich länger für die Berechnung eines Zugs benötigt.

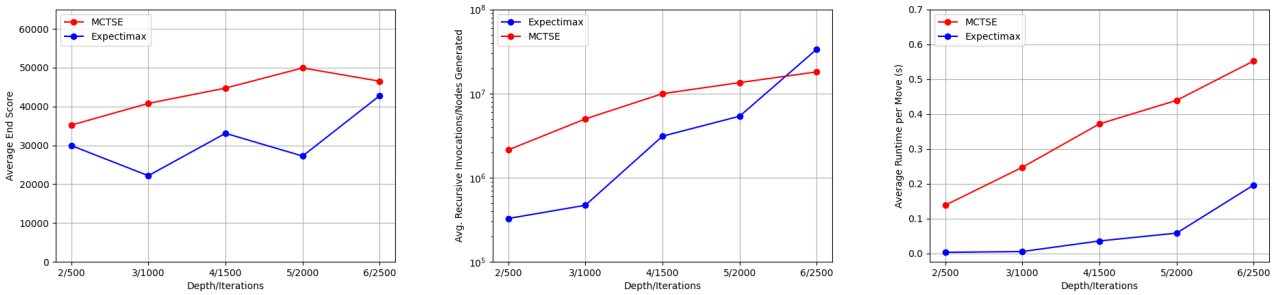


Abbildung 3.7: MCTSE mit Iterationsgrenze vs. Expectimax: Links - erreichter Punktestand, Mitte - Anzahl rekursiver Aufrufe bzw. generierter Knoten und Rechts - benötigte Laufzeit pro Zug

Nun werden die Resultate des MCTSE-Agenten mit angeglichenem Zeit-Budget besprochen und mit Expectimax verglichen. Die Notation ist hier etwas anders: Mit MCTSE$_n$, $n \in \{2, ..., 6\}$ wird der Agent bezeichnet, dessen Zeit-Limit für die Berechnung eines Zuges der durchschnittlichen Laufzeit des korrespondierenden EXP$_n$-Agenten entspricht. In Tab. 3.5 sind diese Laufzeiten dargestellt. Diese entsprechen den numerischen Werten für den blauen Graphen in Abb. 3.3.

| Depth | Avg. Runtime |
|---|---|
| 2 | 0,003 |
| 3 | 0,005 |
| 4 | 0,035 |
| 5 | 0,058 |
| 6 | 0,2 |

Tabelle 3.5: Durchschnittliche Laufzeit der Expectimax-Agenten mit unterschiedlichen Tiefenlimits

### 3.4.2 Erreichte Kacheln

Anhand Tab. 3.6 ist festzustellen, dass der Agent seltener Kacheln mit hohen Werten als Expectimax erreicht (siehe Tab. 3.4). Beispielsweise erzielt MCTSE$_2$ in lediglich 32% der Fälle die 1024-Kachel und in keiner Simulation die 2048-Kachel. Etwas besser performt MCTSE$_3$. Dieser schafft in 68% der Fälle die 1024-Kachel und in jedem zehnten Spiel die 2048-Kachel. Eine deutliche Verbesserung ist bei MCTSE$_4$ und MCTSE$_5$ festzustellen, denn diese erzielen bereits in 74% bzw. 76% der Fälle die 2048-Kachel. MCTSE$_6$ zeigt eine ähnliche Performance wie EXP$_6$ auf, da dieser in fast jedem Durchlauf die 2048-Kachel und in 38% der Spiele die 4096-Kachel erzielte.

| Runtime-Budget | Max Tile | Avg. Max Tile | 4096 [%] | 2048 [%] | 1024 [%] | 512 [%] |
|---|---|---|---|---|---|---|
| 0,003 | 1028 | 660 | 0% | 0% | 32% | 94% |
| 0,005 | 2048 | 963 | 0% | 10% | 68% | 100% |
| 0,035 | 4096 | 1853 | 4% | 74% | 98% | 100% |
| 0,058 | 4096 | 2120 | 16% | 76% | 98% | 100% |
| 0,2 | 4096 | 2785 | 38% | 96% | 100% | 100% |

Tabelle 3.6: MCTSE mit Zeit-Limit - Erreichte Kacheln

In der ersten Gegenüberstellung zeigt sich, dass $EXP_2$, $EXP_3$ und $EXP_4$ besser performen als die entsprechenden MCTSE-Agenten, $MCTSE_5$ und $MCTSE_6$ aber bessere Resultate erzielen.

### 3.4.3 Punktestand

Die Resultate aus dem vorherigen Abschnitt spiegeln sich auch in Abb. 3.8 wider. Diese veranschaulicht die durchschnittlich erreichten Punktestände beider Algorithmen. Erkennbar ist, dass das wachsende Zeit-Budget des MCTSE-Algorithmus mit einem steigenden Punktestand korrespondiert. Bei dem Expectimax-Algorithmus dagegen gibt es bei Tiefe 3 und 5 Einbrüche des Punktestandes. Ferner erzielten die ersten drei MCTSE-Agenten im Mittel eine geringere Punktzahl als die entsprechenden EXP-Agenten, während $MCTSE_5$ und $MCTSE_6$ bessere Punktstände erreichten. Im Konkreten ist $MCTSE_2$ schlechtester Agent mit einem durchschnittlichen Endpunktestand von 8.300 im Gegensatz zum schwächsten $EXP_3$-Agenten, der im Mittel 22.200 Punkte erreichte. Der Punkteunterschied zwischen den jeweils besten Agenten fällt geringer aus. Hierbei erreichte $MCTSE_6$ im Mittel 46.600 Punkte, wohingegen $EXP_6$ 42.800 Punkte erzielen konnte.

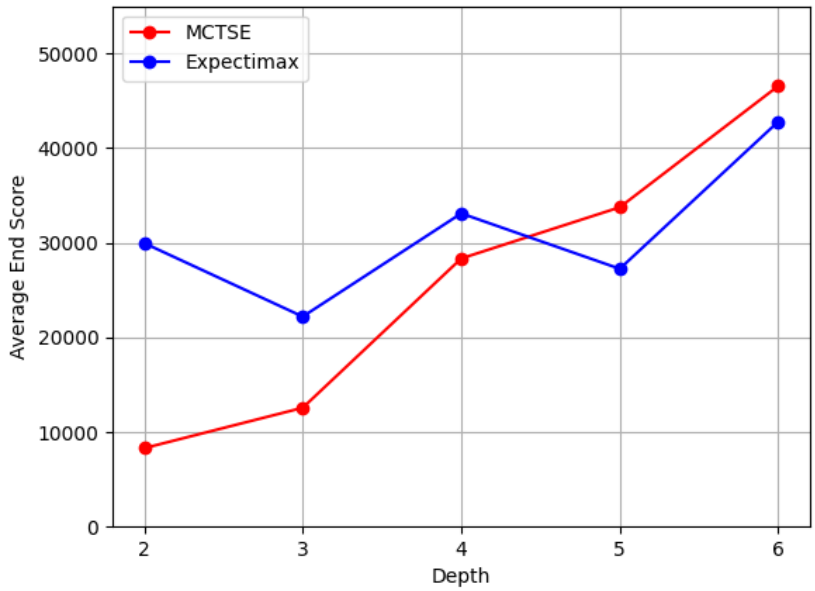


Abbildung 3.8: MCTSE vs. Expectimax: Durchschnittlicher Punktestand am Ende eines Spiels

### 3.4.4 Generierte Knoten vs. Rekursive Aufrufe

Im Folgenden wird die Anzahl generierter Knoten als Synonym für die Anzahl der rekursiven Aufrufe des Expectimax-Algorithmus verwendet. In Abb. 3.9 ist die Anzahl generierter Knoten für den Expectimax- und MCTSE-Algorithmus dargestellt. Diese steigen stetig mit zunehmendem Tiefenlimit bzw. Zeit-Budget und folgen einem ähnlichen Trend. Die konkrete Anzahl generierter Knoten ist jedoch sehr unterschiedlich. $\text{MCTSE}_2$ erzeugte durchschnittlich 19.600 Knoten, während $\text{EXP}_2$ im Mittel 328.000 Knoten erzeugte. Der $\text{MCTSE}_3$-Agent generierte 48.000 Knoten im Mittel, wohingegen $\text{EXP}_3$ ungefähr das 10-Fache erzeugte. Zusätzlich generierten $\text{EXP}_4$ und $\text{EXP}_5$ um etwa eine halbe Größenordnung mehr Knoten als die entsprechenden MCTSE-Agenten. Und schlussendlich generierte der EXP-Agent mit dem größten Tiefenlimit etwa 33,6 Mio. und $\text{EXP}_6$ ungefähr 5,2 Mio. Knoten.

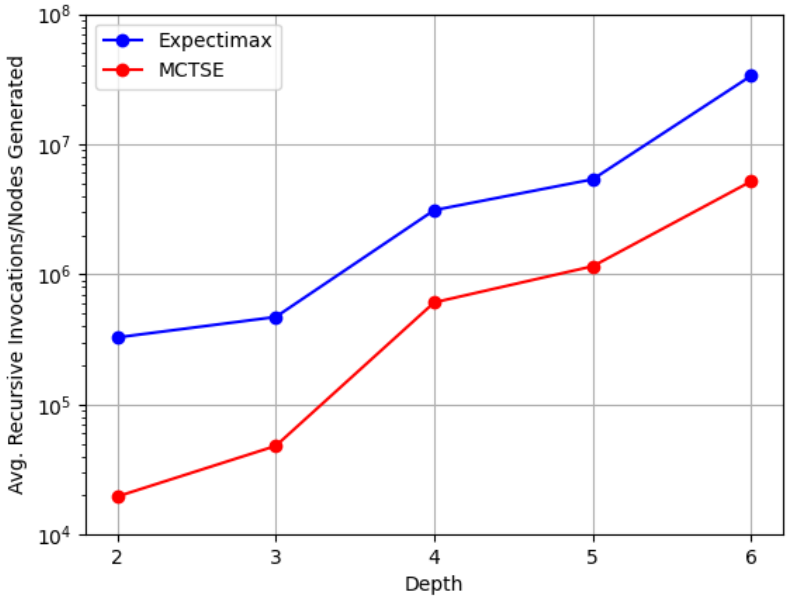


Abbildung 3.9: MCTSE vs. Expectimax: Durchschnittliche Anzahl rekursiver Aufrufe bzw. generierter Knoten in einem Spiel

# 4 Diskussion

## 4.1 Expectimax-Suche

### 4.1.1 Tiefenlimit

Die Simulationsergebnisse deuten mehrheitlich darauf hin, dass je höher das Tiefenlimit, desto besser performt Expectimax. Überraschend ist, dass die Agenten $EXP_3$ und $EXP_5$ mit ungeradem Tiefenlimit schlechter performen, als $EXP_2$ und $EXP_4$, obwohl diese deutlich größere Suchbäume aufbauen und damit mehr mögliche Zukunftsszenarien in Erwägung ziehen. Bei einem ungeraden Tiefenlimit erzeugt die CHANCENODE-Funktion alle möglichen Folgezustände, die durch das Erscheinen einer 2- oder 4-Kachel auf einem leeren Feld entstehen. Danach wird die MAXNODE-Funktion aufgerufen und direkt der neue Brettzustand evaluiert, falls noch ein valider Zug möglich ist. Bei einem geraden Tiefenlimit dagegen werden beim Erreichen dessen Zustände bewertet, die durch das Ausführen der möglichen Aktionen entstehen. Somit enthält ein Suchbaum mit gerader Tiefe deutlich weniger Blattknoten als einer mit ungerader. Dies führt vermutlich dazu, dass ausgehend vom Wurzelknoten eine Aktion ausgewählt wird, die unvorteilhaft ist, da zu viele ähnliche Blattknoten das Ergebnis verfälschen und damit die Gesamtperformance des Agenten darunter leidet.

Im Vergleich zu anderen in der Literatur vorgestellten Agenten erzielte unser Agent mit den untersuchten Tiefenlimits vergleichbare Ergebnisse. Beispielsweise erzielte Neller's Expectimax-Agent für das Tiefenlimit 2 durchschnittlich 14.241 und für das Tiefenlimit 3 25.270 Punkte [9]. Unser $EXP_2$-Agent konnte im Vergleich etwa 30.000 Punkte erzielen und ist damit sogar besser als Neller's Agent mit Tiefenlimit 3. $EXP_3$ erzielte jedoch lediglich 22.194 Punkte und ist damit schlechter als der entsprechende Agent von Neller.

Stünden größere Rechenressourcen zur Verfügung, wäre es interessant gewesen zu überprüfen, wie der Agent mit einem Tiefenlimit größer als 6 agiert. Vor allem um zu überprüfen, ob das zuvor erwähnte Verhalten der Agenten mit ungeradem Tiefenlimit weiterhin bestehen bleibt oder ein Agent die 8192-Kachel erreicht. Neben dem Festlegen eines Tiefenlimits vor dem Ausführen des Algorithmus wäre es ebenso interessant gewesen zu testen, wie der Agent sich mit einem dynamisch ermittelten Tiefenlimit verhält. Dieses könnte beispielsweise niedrig gewählt werden, falls viele freie Felder auf dem Spielbrett vorhanden sind und daher ein schlechter Zug keinen erheblichen negativen Einfluss auf zukünftige Spielstellungen hätte. Oder dieser wird entsprechend hoch gewählt, falls wenige freie Felder auf dem Spielfeld sind und damit mehr Zukunftsszenarien in Erwägung gezogen werden müssen, um keinen ungünstigen Zug zu tätigen.

### 4.1.2 Heuristik- und Parameterwahl

Die Auswahl der Heuristiken für die Evaluationsfunktion hat den größten Einfluss auf die Performance des Expectimax-Algorithmus. Dabei ist einerseits wichtig, dass diese wünschenswerte Kachelanordnungen kodieren. Und andererseits sollen diese effizient und ressourcenschonend zu implementieren sein. In dieser Arbeit wurden drei intuitive und leicht zu berechnende Heuristiken, nämlich *Monotonicity, Smoothness* und *Empty-Cells* vorgestellt. Für diese wurden drei feste Parameter gewählt, die am geeignetsten schienen und die insgesamt vorteilhafte Kachelanordnungen ermöglicht haben. Im Vergleich zu dem besten Expectimax-Agenten von Yarasca und Nguyen, der die gleichen Heuristiken verwendet und einen durchschnittlichen Punktestand von 13.410 erreicht [14], erzielt unser Agent deutlich bessere Ergebnisse.
Andere Ansätze, um optimale Gewichte für die verwendeten Heuristiken zu finden, sind mithilfe von neuronalen Netzen oder durch evolutionäre Algorithmen. Letzteren Ansatz verfolgte Kutsch für seinen MCTSE-Agenten und Xiao für seinen Expectimax-Agenten, um optimale Gewichte zu finden [8][13]. Um die Performance weiter zu verbessern, können daher einerseits die Gewichte optimiert werden und zum anderen ist es durchaus möglich, eine Performanceverbesserung durch eine andere Kombination der Heuristiken mit den Gewichten zu erreichen, als in dieser Arbeit vorgestellt. Mit diesen Anpassungen wäre es durchaus möglich, die 8192-Kachel zu erreichen, vor allem aus dem Grund, da $\text{EXP}_6$ dieser in einer Simulation sehr nahe kam.

### 4.1.3 Transpositionstabelle

Der Einsatz einer Transpositionstabelle für das Speichern bereits besuchter Zustände hat eine erhebliche Geschwindigkeitssteigerung ermöglicht. Unser Agent konnte im extremsten Fall die Laufzeit von $10\frac{s}{Zug}$ beim $\text{EXP}_6$ ohne TT auf $0,2\frac{s}{Zug}$ beim Agenten mit TT verbessern und ist damit um den Faktor 50 schneller. Darüber hinaus wurde auch die Anzahl der rekursiven Aufrufe für das Tiefenlimit 6 von durchschnittlich 960 Mio. auf 33,6 Mio. reduziert und dadurch der verwendete Stack-Speicherverbrauch deutlich verringert. Im Kontrast dazu ist die Größe und damit der Speicherverbrauch der Transpositionstabelle hoch, da für jeden neu besuchten Zustand die aktuelle Tiefe und der Utility-Wert gespeichert wird.

Auch der Expectimax-Agent von Xiao verwendet neben einer effizienten 64-Bit Integer-Darstellung für das Spielbrett eine Transpositionstabelle, um effizient Züge zu berechnen und benötigt für ein maximales Tiefenlimit von 8, das dynamisch bestimmt wird zwischen $0,01$ und $0,2\frac{s}{Zug}$. Unsere Implementierung benötigte im Vergleich zwischen $0,003$ und $0,2\frac{s}{Zug}$ und konnte lediglich deutlich geringere Kacheln erzielen, denn Xiaos Agent schaffte es in jedem dritten Spiel, die 32768-Kachel zu erzeugen [13].

## 4.2 MCTS und MCTSE

### 4.2.1 Iterationsgrenze

Die Iterationsgrenze hat einen signifikanten Einfluss auf die Erfolgsrate der Agenten. Je höher diese ist, desto besser performt in der Regel der Agent, da dieser durch vermehrtes Sampling in der Simulationsphase abschätzt, wie erfolgreich bestimmte Aktionen in der Zukunft sein könnten. In dieser Arbeit wurden sowohl für die Standardvariante als auch für die optimierte fünf Iterationsgrenzen verwendet, die realisierbar schienen. Wie zu erwarten, steigt die Anzahl der Knoten proportional zur Iterationsgrenze. Dabei erzeugte MCTSE deutlich mehr Knoten als MCTS. Dies lag an den zusätzlichen Zufallsknoten, die das Erscheinen einer zufälligen Kachel modellierten. Überdies weisen beide Algorithmen eine ähnliche Laufzeit auf, wobei MCTSE im Mittel etwas länger für einen Zug benötigt.

Stünden größere Rechenressourcen zur Verfügung, wäre es interessant zu testen, welche Resultate der Agent bei deutlich höheren Iterationen erzielt. Der MCTSE-Agent von Kutsch wurde beispielsweise 50x für 3500 Iterationen simuliert und konnte durchschnittlich mehr als 50.000 Punkte erreichen [8]. Unser bester MCTSE-Agent konnte vergleichsweise mit 50.000 Punkten ebenfalls einen soliden Punktestand erreichen, trotz deutlich weniger Iterationen pro Zug. Zudem konnte unser MCTS-Agent bessere Ergebnisse als der MCTS-Agent von Yarasca und Nguyen erzielen [14].

### 4.2.2 Selektionsstrategie

Für MCTS(E) wurde der standard UCT-Algorithmus verwendet in Kombination mit der Explorationskonstante $C = \sqrt{2}$. Diese erzielte die besten Ergebnisse, ist aber im Allgemeinen spielabhängig. Neben dem klassischen UCT-Ansatz gibt es diverse Variationen wie UCB1-Tuned [15] oder UCTN, den Kutsch mit UCT verglichen hat und der leicht bessere Resultate erzielen konnte [8]. Des Weiteren wäre eine nähere Betrachtung von MCTS-Varianten für Einzelspieler-Spiele wie SP-MCTS interessant gewesen, die einen dritten Term im UCT-Algorithmus einführt [3].

### 4.2.3 Rollouttiefe und -strategien

In dieser Arbeit wurde 8 als Rollout-Tiefe gewählt. Dies aus dem Grund, da einerseits die Rechenressourcen sehr begrenzt waren und zum anderen diese eine gute Balance zwischen Rechentiefe und Geschwindigkeit schien. Die meisten Agenten von Kutsch wurden dagegen mit einer Rollout-Tiefe von 150 simuliert [8]. Eine so hohe Rollout-Tiefe scheint weniger praktikabel, da der Speicher- und Rechenaufwand nicht im Verhältnis zum tatsächlichen Nutzen der generierten Daten steht. MCTS basiert immerhin auf der Idee, möglichst viele Simulationspfade in relativ kurzer Zeit zu evaluieren, um fundierte Entscheidungen treffen zu können.
Als Rollout-Strategie wurde die Random-Policy gewählt und zum finalen Bewerten eines Zustands am Ende der Simulationsphase wurde statt der Evaluationsfunktion, die für Expectimax zum Einsatz kam, lediglich die Empty-Cells-Heuristik genutzt, da diese die besten Ergebnisse erzielte. Darüber hinaus sind auch Ansätze möglich, die Heuristiken in der Simulation einbinden [15]. Beispielsweise hat Kutsch für einen Agenten in der Simulationsphase den Rollout komplett durch eine Bewertung des Zustands durch verschiedene Heuristiken ersetzt, darunter auch die in dieser Arbeit vorgestellten Heuristiken. Mit diesen, in Kombination mit optimierten Gewichten, gelang es dem besten Agenten,

durchschnittlich 60.000 Punkte zu erreichen [8].
Ferner besteht die Möglichkeit, die Simulationsphase zu parallelisieren. Dies beschleunigt zum einen die Berechnungen und erhöht zum anderen die Skalierbarkeit.

## 4.3 MCTSE vs. Expectimax

Einerseits wurde die Performance des MCTSE-Algorithmus für unterschiedliche Iterationsgrenzen untersucht. Dabei stellte sich heraus, dass dieser wesentlich konsistenter die 2048- bzw. 4096-Kachel erreichte, als der Expectimax-Algorithmus für die untersuchten Tiefenlimits. Zum anderen wurde die Laufzeit des MCTSE-Algorithmus angeglichen, um einen fairen Vergleich zwischen beiden Algorithmen zu ermöglichen. Bei diesem Vergleich zeigte sich, dass Expectimax zwar für geringe Tiefenlimits besser ist, falls MCTSE jedoch ein größeres Zeit-Limit zur Verfügung hat, erzeugt dieser etwas bessere Resultate. Betrachtet man die Simulationsergebnisse von Yarasca und Nguyen, verwenden diese, wie Kutsch[8], hohe Iterationsgrenzen für MCTS und erzielen häufiger hohe Kachelwerte mit soliden Punkteständen, im Vergleich zu Expectimax. Diese weisen aber im Kontrast eine längere Laufzeit auf [14]. Unsere Ergebnisse weisen eine vergleichbare Tendenz auf.

## 4.4 Limitation dieser Arbeit und Reproduzierbarkeit

Im Umfang dieser Arbeit wurden für jeden vorgestellten Agenten genau 50 Simulationen durchgeführt. Um aussagekräftigere Daten zu erzielen und bessere Aussagen über die Generalisierbarkeit der vorgestellten Methoden zu treffen, wäre es angemessen gewesen, mindestens 100 Simulationen für jeden Agenten durchzuführen. Darüber hinaus hätte man testen können, wie die Agenten für weitere Tiefenlimits bzw. Iterationsgrenzen/Zeit-Limits abgeschnitten hätten. Diese genannten Aspekte waren, wie bereits erwähnt, aufgrund von Ressourcenknappheit nicht realisierbar. Ferner liefert der Vergleich zwischen dem optimierten Expectimax- und MCTSE-Algorithmus keine Aussage darüber, welcher Algorithmus im Allgemeinen besser für das Lösen des Puzzles geeignet ist, sondern präsentiert vielmehr, wie diese für festgelegte Parameter und Strategien bzw. Heuristiken abschneiden. Um solch eine Aussage ansatzweise treffen zu können, müssten mehr Optimierungsmöglichkeiten und Parameterkombinationen in Erwägung gezogen und verglichen werden. Stattdessen konnte gezeigt werden, dass Expectimax mit Transpositionstabelle einen deutlichen Geschwindigkeitsvorteil bringt und dass MCTSE besser für 2048 performt, als die klassische Variante. Letzteres bestätigt damit die Experimente von Kutsch [8].

# 5 Fazit

Wie in der Einleitung erwähnt, war das primäre Ziel dieser Arbeit, das 2048-Puzzle mithilfe der vorgestellten Verfahren mit höchster Wahrscheinlichkeit algorithmisch zu lösen. Sekundär sollte ein Vergleich beider optimierten Verfahren in Kombination mit domänenspezifischen Heuristiken durchgeführt werden, um zu ermitteln, welche besser für das Lösen des Puzzles geeignet sind. Im Folgenden werden die Ergebnisse kurz zusammengefasst und besprochen, welche Schlüsse gezogen werden können.

- Der Expectimax-Algorithmus konnte das Puzzle lösen und darüber hinaus in zahlreichen Durchläufen die 4096-Kachel erreichen. Das Einsetzen einer Transpositionstabelle zum Speichern besuchter Zustände ist durchaus sinnvoll, denn es ist eine erhebliche Geschwindigkeitssteigerung gegenüber der klassischen Variante festzustellen. Überdies impliziert ein höheres Tiefenlimit nicht unbedingt bessere Ergebnisse und die Anzahl rekursiver Aufrufe wächst exponentiell mit dem Tiefenlimit. Zudem ist die Wahl adäquater Heuristiken und Gewichte essenziell für das zielgerichtete Lösen des Puzzles. Dabei hat sich herausgestellt, dass Heuristiken, die intuitive Herangehensweisen beim Lösen des Puzzles modellieren, besonders gut geeignet sind.

- Beide MCTS Varianten konnten solide Ergebnisse erzielen. Die optimierte Version hat sich durch konsistentes Erreichen der 2048-Kachel ausgezeichnet und der beste Agent war in der Lage in jedem zweiten Spiel die 4096-Kachel zu erzielen. Als Evaluationsfunktion wurde die effiziente Empty-Cells-Heuristik verwendet, die sich als einfach und erfolgreich herausstellte. MCTSE hat im Vergleich zu MCTS bessere Resultate erzielt, mit erhöhten Speicherverbrauch, aber ähnlicher linearer Laufzeit. Auch hier impliziert eine höhere Iterationsgrenze nicht bessere Resultate. Ferner wurde MCTSE für unterschiedliche Zeit-Limits getestet. $\text{MCTSE}_{2,3}$ zeigten mangelhafte Resultate. $\text{MCTSE}_{4,5,6}$ lösten das Spiel bereits deutlich häufiger, wobei $\text{MCTSE}_{6}$ in über einem Drittel die 4096-Kachel erzielen konnte.

- Die optimierten Versionen wurden anhand festgelegter Kriterien miteinander verglichen. Dabei performten $\text{EXP}_{2,3,4}$ besser als die entsprechenden MCTSE-Agenten, während $\text{MCTSE}_{5,6}$ für ein größeres Zeit-Budget das Spiel mit einer leicht höheren Wahrscheinlichkeit lösen. Folglich legen die Resultate nahe, dass MCTSE für ein höheres Zeit-Budget das Spiel mit einer höheren Wahrscheinlichkeit und geringerer Knotengenerierung lösen kann, als Expectimax.

Mit diesen Resultaten kann geschlussfolgert werden, dass die untersuchten Algorithmen für die gewählten Parameter geeignete Verfahren sind, um das 2048-Puzzle zu lösen.
Für zukünftige Forschungen wäre es interessant zu untersuchen, wie Expectimax für optimierte Parameter und höheres Tiefenlimit abschneidet und welche weiteren Heuristiken mit dem Spiel kompatibel sind. Bei MCTSE wäre es interessant zu analysieren, wie dieser, mit einer höheren Iterationsgrenze bzw. Zeit-Budget, anderen Heuristiken, größeren Rollouttiefen und verschiedenen UCT-Varianten abschneidet.

# Literaturverzeichnis


[1] Transposition Table. https://www.chessprogramming.org/Transposition_Table#How_it_works. Abgerufen: 08.10.24.

[2] Zobrist Hashing. https://www.chessprogramming.org/Zobrist_Hashing. Abgerufen: 08.10.24.

[3] Cameron B. Browne, Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[4] Gabriele Cirulli. 2048. https://play2048.co/, 2014.

[5] Hung Guei, Lung-Pin Chen, and I-Chen Wu. Optimistic Temporal Difference Learning for 2048. *IEEE Transactions on Games*, 14(3):478–487, 2022.

[6] Wojciech Jaśkowski. Mastering 2048 With Delayed Temporal Coherence Learning, Multistage Weight Promotion, Redundant Encoding, and Carousel Shaping. *IEEE Transactions on Games*, 10(1):3–14, 2018.

[7] Iris Kohler, Theresa Migler, and Foaad Khosmood. Composition of Basic Heuristics for the Game 2048. In *Proceedings of the 14th International Conference on the Foundations of Digital Games*, FDG '19, New York, NY, USA, 2019. Association for Computing Machinery.

[8] Johannes Kutsch. KI-Agenten für das Spiel 2048: Untersuchung von Lernalgorithmen für nichtdeterministische Spiele, 2017. Bachelorarbeit, Technische Hochschule Köln.

[9] Todd W. Neller. Pedagogical Possibilities for the 2048 Puzzle Game. In *The Journal of Computing Sciences in Colleges 30.3*, pages 38–46, 2015.

[10] Phillip Rodgers and John Levine. An Investigation into 2048 AI Strategies. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–2, 2014.

[11] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 2016.

[12] Marcin Szubert and Wojciech Jaśkowski. Temporal Difference Learning of N-tuple Networks for the Game 2048. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8, 2014.

[13] Robert Xiao. Expectimax Optimization for 2048. https://stackoverflow.com/a/22498940, 2014. Abgerufen: 03.11.24.

[14] Efrain Noa Yarasca and khoi Nguyen. Comparison of Expectimax and Monte Carlo algorithms in solving the online 2048 game. *PESQUIMAT*, 21(1):1–10, 2018.

[15] Maciej Świechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mańdziuk. Monte Carlo Tree Search: a review of recent modifications and applications. *Artificial Intelligence Review*, 56(1):2497–2562, 2023.