An:
Christian Thalmann und
Anh Huy Truong
MNG Rämibühl
Rämistraße 58
8001 Zürich

Vergleich der Komplexität von Tree-Search-Algorithmen

Bericht zum PU-Projekt
von Fabian Stämpfli und Djordje Petrović (4h)
Kantonsschule MNG Räibühl

Zürich, 21.04.2023

Abstract

In diesem Projekt wird die Komplexität von verschiedenen einfachen Tree-Search-Algorithmen untersucht. Es werden die Tiefensuche, Breitensuche und Monte-Carlo Baumsuche auf binären Suchbäumen getestet. Dabei wird die Anzahl der durchsuchten Knoten und die benötigte Zeit gemessen, um die Effizienz der Algorithmen zu vergleichen. Das Ziel ist es, die Komplexität der Suchalgorithmen zu berechnen, um diese dadurch zu vergleichen.

Inhaltsverzeichnis

1 Einleitung

Das Ziel dieses Projekts ist es, die Komplexitäten verschiedener Tree-Search-Algorithmen hinsichtlich der Größe und Tiefe des Baumes zu vergleichen. Dabei wird ein Programm in der Programmiersprache C++ erstellt, um die Berechnungen zu messen und zu überprüfen.

Tree-Search-Algorithmen sind in der künstlichen Intelligenz weit verbreitet und werden verwendet, um Entscheidungsprobleme zu lösen. Da verschiedene Algorithmen unterschiedliche Vorteile und Nachteile aufweisen, ist es wichtig, ihre Leistungsfähigkeit und Komplexität zu verstehen. Eine Art, dies zu messen ist die Einschätzung der Komplexität. Diese ist eine Abschätzung für das obere Limit der Laufzeit eines Tree-Search-Algorithmus bei einer Anzahl von n und einer Tiefe von d Knoten. In diesem Projekt wird die Komplexität von verschiedenen Tree-Search-Algorithmen miteinander verglichen, um eine bessere Einschätzung der Komplexität bei der Verwendung dieser Algorithmen zu ermöglichen.

2 Theorie

2.1 Baum

Suchbäume sind Datenstrukturen aus der Graphentheorie. Der Begriff 'Suchbaum' bezieht sich darauf, dass die Datenstruktur baumartig organisiert ist. Ein Baum besteht aus einer Wurzel, die den obersten Knoten (engl. nodes) darstellt, und einer Anzahl von Zweigen oder Kanten, die von der Wurzel zu den Blättern führen, die die Endpunkte des Baums darstellen. Jeder Knoten im Baum repräsentiert eine Entscheidung oder einen Zustand, und jeder Zweig repräsentiert eine Aktion, die von diesem Zustand aus ausgeführt werden kann.

Ein Baum ist also ein zusammenhängender Graph, der keine geschlossenen Pfade enthält (Diestel, 1996), d.h. der sich nur in eine Richtung von der Wurzel aus erstreckt.

2.1.1 binärer Baum

EIn binärer Baum ist eine besondere Art von Baum, bei dem jeder Knoten maximal zwei Kindknoten (engl. *child nodes*) haben kann. Ein solcher Baum ermöglicht eine effizientere Suche, da Suchalgorithmen sich an der Baumstruktur orientieren können, um die Suche zu optimieren und dadurch schneller Ergebnisse zu liefern.

Ein binärer Baum kann ausgeglichen/balanciert, teilweise balanciert oder unbalanciert sein. Wenn er ausgeglichen ist, bedeutet dies, dass beide Teilbäume eines Knotens (links und rechts) immer gleich Tief sind, während bei unbalancierten Bäumen die Tiefe der Teilbäme sich beliebig unterschieden kann. Bei einem unbalancierten binären Baum kann zum Beispiel ein Teilbaum vollständig leer sein, während beim anderen alle Knoten sind. Ein teilweise balancierter Baum überschreitet eine gewisse Tiefendifferenz der Teilbäme nicht.

2.1.2 binärer Suchbaum

Bei einem binärer Suchbaum kann jeder Knoten nur zwei Kindknoten haben. Ausserdem erfordert ein binärer Suchbaum, im Gegensatz zu einem normalen Suchbaum, eine bestimmte Anordnung der Knoten: Diese sind so angeordnet, dass jeder Knoten kleiner als alle Knoten im rechten Teilbaum ist, aber grösser als alle Knoten im linken Teilbaum ist (Goodrich, T., Tamassia & Mount, 2011).

Der Vorteil dieser speziellen Anordnung ist, dass man schnell einen Knoten im Baum finden kann, da mit jedem durchlaufenen Knoten der Suchraum, also der Bereich, in dem nach einem bestimmten Knoten gesucht wird, durch das Vergleichen von Schlüsselwerten immer weiter eingeschränkt und halbiert werden kann. Auch ist das Einfügen und Löschen von Knoten aus dem Baum einfacher.

2.2 Tree-Search Algorithmus

Tree-Search-Algorithmen sind eine Art von Algorithmen, die zur Lösung von Suchproblemen verwendet werden. Sie sind nützlich, wenn aus einer grossen Anzahl von möglichen Optionen die passende gefunden werden muss.

Tree-Search-Algorithmen arbeiten mit Suchbäumen, in denen die verschiedenen Optionen als Knoten, die über Kanten verbunden dargestellt sind. Die Suche nach der Lösung des Problems besteht darin, durch die Baumstruktur zu navigieren und den Pfad zu finden, der von der Wurzel (engl. root) bis zum Zielzustand führt.

2.3 Komplexität von Tree-Search-Algorithmen

Das Durchsuchen eines Baumes nach einem gewünschten Zustand dauert nicht immer gleich lange. Es hängt vielmehr von einer Vielzahl verschiedener Faktoren ab, wie zum Beispiel der Größe und Tiefe des Suchbaumes oder der Anzahl der Knoten. Eine Abschätzung für die obere Schranke der Suchdauer und damit der Effizienz eines Tree-Search-Algorithmus gibt die Komplexität.

Die Komplexität von Tree-Search-Algorithmen wird in der sogenannten O-Notation ausgedrückt, die angibt, wie schnell die Laufzeit eines Suchalgorithmus mit der Grösse eines Baumes wächst. Eine Komplexität von $O(n)$ bedeutet beispielsweise, dass die Laufzeit eines Algorithmus linear von der Anzahl der Knoten n abhängt.

3 Algorithmen im Vergleich

Untersucht wurden verschiedene Tree-Search-Algorithmen, unter anderem die Tiefensuche (engl. depth-first search, DFS), die Breitensuche (engl. breadth-first search, BFS) und die Monte-Carlo-Baumsuche (engl. Monte Carlo tree search, MCTS). Die Komplexität wurde zunächst für jeden Suchalgorithmus berechnet und anschliessend mit dem Programm in Bezug auf die Anzahl der Baumknoten und die Tiefe, in der das Ergebins lag, überprüft.

3.1 Der Suchbaum

Das Projekt wurde mit einem binären Suchbaum umgesetzt, da diese Baumart den Vorteil hat, leicht skalierbar zu sein. Auch ist die Tiefe des Baumes logarithmisch zu der Anzahl der Knoten, was die Regelung der Baumtiefe erleichtert. Da jeder Knoten zwei *child*-Knoten enthält, ist die Tiefe d eines binären Baumes mit n Knoten

$$d \approx \log_2(n)$$

(Goodrich et al., 2011).

Der Baum wurde wie folgt umgesetzt: Der Baum besteht aus einzelnen Knoten, die aus einem *struct*-Datentyp bestehen, der über Pointer auf die *child*- und *parent*-Knoten verweist.

```
struct node {
public:
    int data;
    node* left;
    node* right;
    node* parent;
    node(int d) : data(d), left(NULL), right(NULL), parent(NULL) {}
};
```

Der Vorteil einer solchen Struktur ist, dass einzelne Knoten leicht geändert werden können. Zusätzlich beschleunigt eine solche Implementierung den Aufruf der einzelnen Knoten, was die Gesamtlaufzeit des Programms verkürzt. Messungen haben ergeben, dass bei dieser Umsetzung die Knoten etwa mit

$$O(Knoten) \approx O(1)$$

aufgerufen werden können, daher wurde diese Komplexität für die folgenden Rechnungen benutzt.

3.2 Breitensuche

Die Breitensuche (engl. Breadth-First-Search, BFS) ist ein ein Tree-Search Algorithmus, der Graphen schichtweise durchsucht, beginnend bei einem Startknoten und sich schrittweise auf benachbarte Knoten ausbreitet (Abbildung 1).

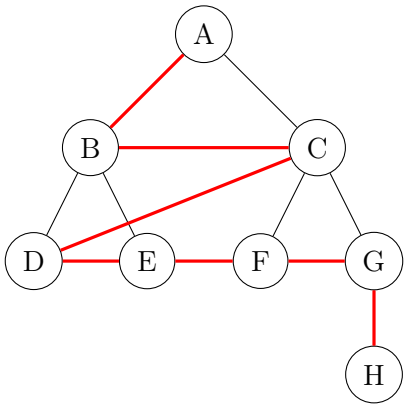

Abbildung 1: Der Breitensuchalgorithmus auf einem binären Baum, rot dargestellt

Der im Programm gebrauchte Algorithmus benutzte eine *Queue*, also eine Warteschlangen-Datenstruktur für die Speicherung des Baumes. Dabei werden jeweils alle Knoten in einer Schicht nacheinander durchsucht, auf *child*-Knoten untersucht und anschliessend in die *Queue* gespeichert. Dies wiederholt sich bis zum Baumende.

Die schichtweise Suche sorgt dafür, dass Knoten in gleicher Entfernung vom Startknoten zuerst besucht werden, bevor Knoten in weiter entfernten Schichten besucht werden. Dies ist besonders effizient bei flachen Bäumen.

3.2.1 Berechnung der Komplexität

Die Komplexität der Breitensuche in einem binären Baum mit Tiefe d und n Knoten beträgt:

$$\text{Komplexität} = \sum_{i=0}^{d} \text{Laufzeit der Ebene } i \tag{1}$$

$$= \sum_{i=0}^{d} O(\text{Anzahl der Knoten in Ebene } i) \tag{2}$$

$$= O(\sum_{i=0}^{d} \text{Anzahl der Knoten in Ebene } i) \tag{3}$$

$$= O(\sum_{i=0}^{d} 2^1) \tag{4}$$

$$= O(2^d) \tag{5}$$

$$= O(n), \tag{6}$$

da die Breitensuche jeden Knoten höchstens einmal speichert. Der Speicherplatzbedarf für die *Queue* beträgt also im schlechtesten Fall, wenn der gesuchte Wert in den tiefsten Blättern ist und alle anderen Knoten zuvor durchgegangen werden müssen

$$O(n)$$

, da die *Queue* alle Knoten speichern muss, die besucht werden.Daher ist die Laufzeit der Breitensuche proportional zur Grösse des Baumes, also zur Anzahl Knoten.

Die Tiefe des Baumes hat auch einen Einfluss auf die Laufzeit, da die Suche mehr Knoten besuchen muss, bevor die Blätter erreicht werden. Da in dieser Umsetzung die Aufrufzeit der Knoten unabhängig von der Tiefe $O(1)$ ist, bleibt die Laufzeit aber immer noch proportional zur Anzahl der Knoten.

Dies wurde jedoch für einen unbalancierten Baum berechnet. Diese Komplexität gilt auch für ausbalancierte Bäume, doch da die Suche einen Knoten abbricht, sobald er gefunden wurde, ist die durchschnittliche Komplexität in einem ausbalancierten Baum kleiner, da weniger leere Knoten durchlaufen werden müssen. Dann der Durchschnitt

$$O(\log_2(n)).$$

3.3 Tiefensuche

Die Tiefensuche (engl. Depth-First-Search, DFS) ist ein Tree-Search Algorithmus der einen Graphen in die Tiefe durchsucht, d.h. sie geht so weit wie möglich in einen Pfad, bevor sie sich umwendet und nach anderen Pfaden sucht (Abbildung 2).

Der für das Projekt geschriebene Algorithmus verwendet eine *stack*-Datenstruktur, um die Reihenfolge der besuchenden Knoten zu speichern. Dabei wird ein Pfad durch den Baum gewählt, der bis zum einem Blatt begangen wird. Danach wird der Algorithmus bis zum letzten nicht vollständig erforschten Knoten zurückgeführt, von wo die Suche nochmals beginnt. Dieser Vorgang wird fortgesetzt, bis alle Knoten besucht wurden.

3.3.1 Berechnung der Komplexität

Die Tiefensuche läuft alle Pfade des Baumes nacheinander in die Tiefe durch. Wenn der Baum nicht ausbalanciert ist, kann es passieren, dass ein Pfad von der Wurzel zu einem

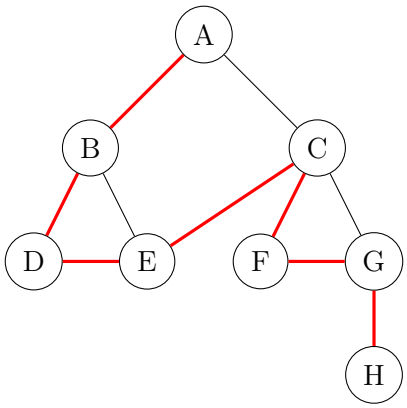

Abbildung 2: Der Tiefensuchalgorithmus auf einem binären Baum, rot dargestellt

Blatt eine Länge von n hat. Dann muss die Suche im schlechtesten Fall alle Knoten durchlaufen und hat damit ebenfalls eine Komplexität von

$$O(n).$$

Die Tiefe ändert auch bei der Tiefensuche die Komplexität nicht, da alle Knoten innerhalb von $O(n)$ aufgerufen werden können.

Wenn der Baum ausbalanciert ist, beträgt die durchschnittliche Komplexität wie bei der Breitensuche

$$O(\log_2(n)),$$

da in allen Teilbäumen gleich viele Knoten sind.

3.4 Monte-Carlo Baumsuche

Die Monte-Carlo Baumsuche ist ein Algorithmus, der verwendet wird, um den besten Pfad in einem Baum von Entscheidungen zu finden. Die Methode basiert auf der Monte-Carlo Methode, die eine große Anzahl von Zufallsexperimenten verwendet, um einen Wert anzunähern. In diesem Fall wird die Methode verwendet, um die Wahrscheinlichkeiten zu bestimmen, mit denen jeder Pfad im Baum durchsucht wird.

Der Algorithmus sucht den Baum von der Wurzel aus, indem er die Wahrscheinlichkeiten jedes Pfades nutzt, um zu entscheiden, welchen Pfad er zuerst durchsuchen soll. Wenn er den gewünschten Wert nicht auf diesem Pfad findet, setzt er seine Suche bei einem anderen Pfad fort. Der Algorithmus arbeitet dabei ähnlich wie eine Tiefensuche, indem er immer bis zum Ende des Baums oder bis zur vorgegebenen Tiefe sucht.

Wenn der Algorithmus keinen passenden Pfad findet, beginnt er seine Suche bei der niedrigsten Tiefe, bei der noch nicht alle Pfade durchsucht wurden. Um die Leistung zu optimieren löscht der für das Projekt geschriebene Algorithmus den zuvor gesuchten Pfad indem er die Suche mit der Funktion *std::async* Funktion.

Der Algorithmus wird verständlicher, wenn man den Code anschaut (nur Hauptfunktion, vereinfacht, ohne Multithreading):

```
1 int nodetest(node* Parent, int Value, int MaxDepth, int Depth) {
2     // Position im Baum initialisieren, zufaelliges Kind initialisieren
3     BT::Position PositionInhit;
4     node* selectedChild = nullptr;
5     // linkes/rechtes zufaellig Kind waehlen, falls es vorhanden ist.
```

```
6       if (Parent->left == nullptr)         selectedChild = Parent->right;
7       else if (Parent->right == nullptr)   selectedChild = Parent->left;
8       else                                 selectedChild = accessNext(Parent)
        ;
9       // Position im Baum aktualisieren
10      PositionInhit.v = selectedChild;
11      // Pruefen, ob das ausgewaehlte Kind ein Blatt ist
12      bool Is_external = PositionInhit.isExternal();
13      // Ist der gesuchte Wert im Kind gefunden worden (success), ein
        externer Knoten oder in der maximalen Tiefe (-1)?
14      if (selectedChild->data == Value)              return selectedChild->
        data;
15      else if (Is_external || Depth == MaxDepth)     return -1;
16      else                                           return nodetest(
        selectedChild, Value, MaxDepth, Depth++);
17  }
```

3.4.1 Berechnung der Komplexität

Die Komplexität des Algorithmus beträgt wie bei den anderen Suchalgorithmen im ungünstigsten Fall $O(n)$, um alle Knoten zu durchlaufen. Dies ist möglich, weil überflüssige Knoten gelöscht werden. Für einen ausgewogenen Baum ist die durchschnittliche Komplexität $O(nI+1)$, wobei I das Produkt aller Gegenwahrscheinlichkeiten ist, dass der richtige Weg zum gesuchten Wert eingeschlagen wird. Da wir in diesem Projekt keine vorgegebenen Wahrscheinlichkeiten für den Baum haben und diese zufällig generiert werden, ist die tatsächliche Komplexität für einen balancierten Baum $O(n/2+1)$. Um die Gesamtzeit zu berechnen, verwenden wir die Formel

$$t(n,l,d) = I * n * (\sum_{k=0}^{n} s^k) + O(m) * I * \sum_{k=0}^{n} s^k,$$

wobei O(m) die Zeitkomplexität des Zufallsgenerators für den Algorithmus und k die Tiefe des Baumes ist. Dies ergibt sich aus folgender Überlegung:

$$\begin{aligned}
\text{Komplexität} &= O(n)\,\text{Zeit pro Knoten} && (7)\\
&= \sum_{i=0}^{d} O(\text{Zeit alle Knoten zu durchlaufen}) && (8)\\
&= \sum_{k=0}^{n} s^k * I * k\,\text{Zeit einen bestimmten Weg zu finden} && (9)\\
&= \sum_{i=0}^{d} *I * k + O(m) * I * \sum_{k=0}^{n} s^k, && (10)
\end{aligned}$$

wobei in (10) $O(m) * I * \sum_{k=0}^{n} s^k$ die Zeit ist, die es braucht, um eine *node* zu selektrieren.

4 Resultate

Bei allen Suchalgorithmen wurde die Laufzeit in Abhängigkeit von der Tiefe des Baumes und der Grösse des Baumes gemessen. Bei der Tiefe wurde ein Baum mit 5′000′000

Knoten benutzt, bei der Grösse Bäume zwischen 1′000 und 5′000′000 Knoten. Alle Messungen wurden 100 Mal durchgeführt, um allfällige Fehler auszugleichen, angezeigt wird jeweils der Durchschnitt der Messungen. Eine grössere Anzahl an Knoten oder Messungen war wegen der begrenzten Rechenleistung der benutzten Geräte nicht möglich; selbst die folgenden Messungen haben jeweils mehr als 45 Minuten gebraucht.

4.1 Breitensuche

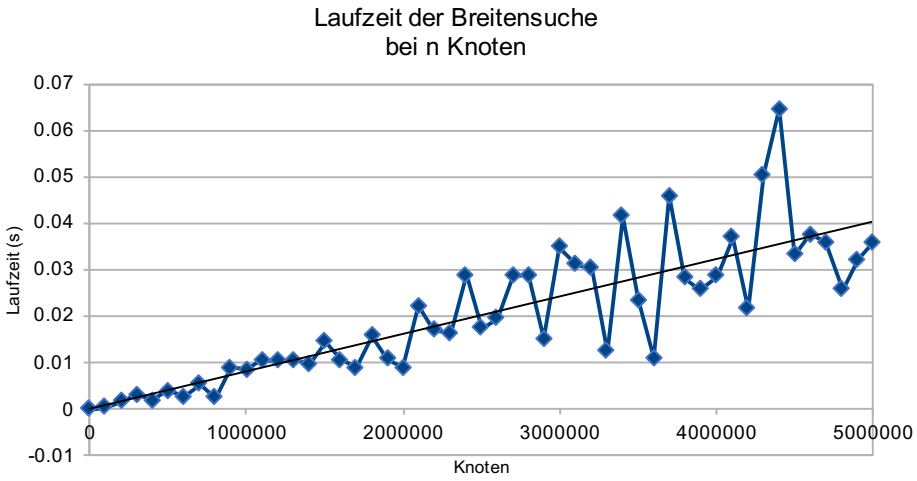

Abbildung 3: Die Laufzeit der Breitensuche, abhängig von der Anzahl Knoten

Die gemessenen Laufzeiten der Breitensuche stimmen ziemlich gut mit der Theorie überein (Abblidung 3): Die Laufzeit steigt linear mit der Anzahl der Knoten, da die gemessenen Bäume jeweils teilweise balanciert sind. Die Ursache der Abweichungen ist die unterschiedliche Balancierung der Bäume.

4.2 Tiefensuche

Die gemessenen Laufzeiten der Tiefensuche stimmen sehr gut mit der Theorie überein, da die Laufzeit linear von der Kontenzahl n abhängt (Abblidung 4). Die grösseren Abweichungen die unterschiedliche Balancierungen der Bäume als Ursache: Aus Zeitgründen wurde für jede Knotenzahl nur ein Baum generiert, in dem 100 Mal unterschiedliche Werte gesucht wurden. Falls ein Baum sehr unbalanciert ist, dauern alle 100 Suchen durchschnittlich länger. Diese Unbalancierung ist bei grösseren Bäumen besser sichtbar als bei kleinen.

4.3 Monte-Carlo Baumsuche

Bei der Monte-Carlo Baumsuche, stimmen die Daten nur teilweise mit der Theorie überein. Zum einen Passt die von Excel erstellte Fitfunktion nicht genau zu den Berechnungen welche eine Fitfunktion von $O(n/I) = 1/2 * O(n)$ vorraussagt. Auch auffällig ist ein starker abfall in der Berechnungszeit von den Nodeanzahlen 170'000 zu 180'000. Dieser ist resultat der zufallsbasierten Pfadentscheidungen des Algorithmus. Da nur eine kleine Menge an Daten aus Zeitgründen errechnet wurden ist dies kaum zu beheben. Die

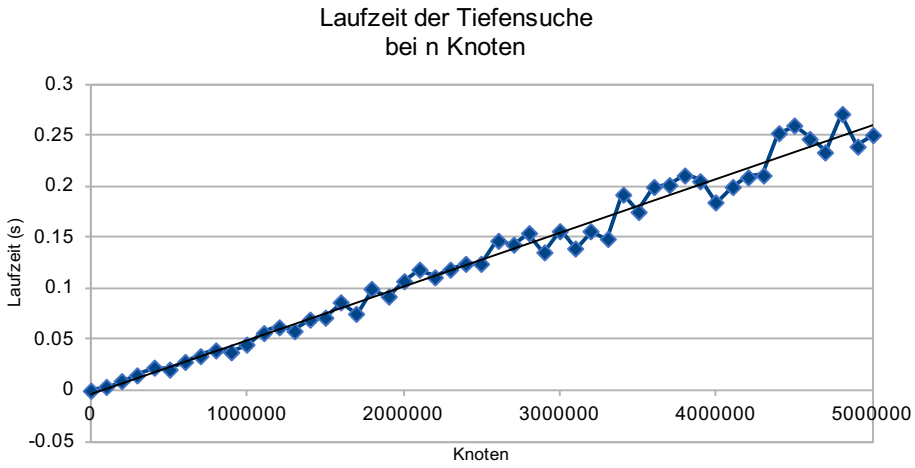

Abbildung 4: Die Laufzeit der Treitensuche, abhängig von der Anzahl Knoten

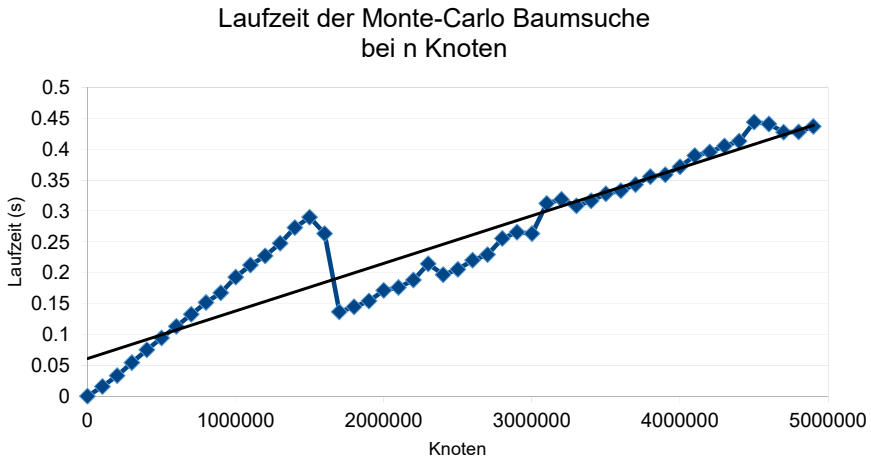

Abbildung 5: Die Laufzeit der Monte-Carlo Baumsuche, abhängig von der Anzahl Knoten

schlechtere Leistung des Algorithmus im Allgemeinen ist mit dem Fakt zu bgeründen, dass der Algorithmus incht für Binäre baume erstellt wurde sondern für Baume die sich viel spalten sogenannte N-Ary Trees und für Bäume bei denen die Warscheinlichkeiten durch beispielsweise ein Neuronales Netzwerk gegeben sind.

5 Schlussfolgerung

Wir haben die Komplexität von verschiedenen Algorithmen im Bezug auf die Tiefe und Grösse eines binären Baumes untersucht. Dazu haben wir die Komplexität der Algorith-

men zuerst berechnet und danach für verschiedene Baumgrössen 100 Mal gemessen.

Die Messungen haben ergeben, dass die Komplexität der drei Algorithmen verschieden sind und jder seine Vorteile hat. Die Monte-Carlo Baumsuche ist sehr gut wenn man kleine Baäume analysiert während tiefen und breitensuche stärker bei grösseren Baumen sind. Die Breiten und die Teifensuche sind auch nochmals unterschiedlich schnell je nach dem wo sich der gesuchte Wert befindet. Da die Werte in diesem Projekt jedoch zufällig platziert wurden, sind kaum geschwindigkeitsunterschiede festzustellen.

6 Reflexion

Das Projekt, vor allem das Schreiben des Programms war ziemlich ansprochsvoll, ging aber gut. Eines der grössten Probleme war die Effizienz des Baumes. Der Code musste mehrmals umgeschrieben werden, um eine schnelle Aufrufezeit der einzelnen Knoten zu ermöglichen.

Als Schwierigkeit gestaltete sich auch das Generieren von Zufallszahlen, die man für die Knotenwerte und das Einfügen der Knoten in den Baum braucht. Zuerst verwendeten wir den Mersenne-Twister Pseudo-Zufallsgenerator, doch dieser war ausserordentlich langsam: um einen Baum mit 100'000 Knoten zu generieren brauchte es über 65 Sekunden. Durch den Einsatz des Xorshift128+ Pseudo-Zufallsgenerators konnte ein Baum gleicher Größe innerhalb von 20 Millisekunden generiert werden - das entspricht einer Geschwindigkeitssteigerung um mehr als das 3'000-fache! Allerdings war dieser Algorithmus nicht Teil der Standardbibliothek von C++- er musste von uns implementiert werden.

Trotz diesen Optimierungen dauerten die Messungen für jeden der Suchalgorithmen mehr als 45 Minuten - insgesamt haben die Rechnungen mehr als 4 Stunden CPU-Zeit beansprucht. Trotzdem sind wir mit dem Projekt sehr zufrieden, da die Ergebnisse sehr gut den Theoriewerten entsprechen.

Der Quellcode für das Programm ist ist öffentlich, Open-Source und kann unter diesem Web-Link aufgerufen werden.

Literatur

Diestel, R. (1996). *Graphentheorie*. Berlin: Springer. print.

Goodrich, T., M., Tamassia, R. & Mount, D. M. (2011). *Data structures and algorithms in c++*. 2nd ed. Hoboken: Wiley. print.