Diplomarbeit

Massiv-parallele Suche und effiziente Heuristiken

Robert Maier
10. Januar 2011

Humboldt-Universität zu Berlin

Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

Gutachter:

Prof. Dr. Alexander Reinefeld
Prof. Dr. Hans-Dieter Burkhard

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit „Massiv-parallele Suche und effiziente Heuristiken“ selbstständig und ohne fremde Hilfe verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Weiterhin erkläre ich hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den 10. Januar 2011

Danksagung

Mein Dank gilt Herrn Prof. Reinefeld und Thorsten Schütt für die vielen konstruktiven Hinweise und Gespräche zu dieser Arbeit und die angenehme Zusammenarbeit der letzten anderthalb Jahre.

Weiterhin möchte ich mich bei meiner Familie und besonders bei Sandra für die fortwährende Unterstützung bedanken.

Kurzfassung

Pattern Datenbanken (PDBs) werden in der heuristischen Suche eingesetzt, um irrelevante Pfade in der Suche auszuschließen. Da die Leistung einer PDB mit ihrer Größe steigt und für die effektive heuristische Suche in großen Zustandsräumen große PDBs nötig sind, ist ein paralleler Ansatz notwendig, um sehr große PDBs zu erzeugen. Diese Arbeit erweitert einen, auf MapReduce basierenden, Algorithmus von Reinefeld und Schütt (2009) zur massiv-parallelen Breitensuche so, dass damit sehr große PDBs erzeugt werden können. So entsteht die erste vollständige 8+8+8 PDB für das 24er-Puzzle. Reinefeld und Schütt implementieren ebenfalls das heuristische Suchverfahren BFIDA* für MapReduce. In dieser Arbeit wird dieser Implementation ein Speedup von 857 bei 2039 Kernen nachgewiesen. Weiterhin führt diese Arbeit ein Schema ein, mit dem PDBs bei der Suche direkt von der Festplatte gelesen werden können.

Des Weiteren wird der Nutzen von großen PDBs in einer Gruppe von PDBs untersucht, deren Heuristikwerte maximiert werden. Dazu wird analysiert, wie sich Gruppen von PDBs mit unterschiedlicher Größe verhalten und welche Faktoren bei solchen Konstellationen zum Erfolg der Gruppe beitragen. Es wird gezeigt, dass der Einsatz von großen PDBs effizient ist, wenn der zur Verfügung stehende Hauptspeicher ausreicht, eine Gruppe zusammen mit einigen kleineren PDBs zu bilden.

Abstract

Pattern Databases (PDBs) are used to exclude irrelevant paths in heuristic search. As the pruning power increases with the size of a PDB and large state spaces require large PDBs for effective heuristic search, a new parallel approach is needed to create very large PDBs. This thesis extends an algorithm of Reinefeld und Schütt (2009) for massively-parallel breadth-first search, enabling it to create PDBs of almost arbitrary size. As a result the first 8+8+8 PDB for the 24-puzzle was created. Reinefeld und Schütt also implemented breadth-first iterative-deepening A* on MapReduce (MR-BFIDA*) which is shown in this paper to achieve a speedup of 857 using 2039 cores. In addition, a scheme is presented which is based on MR-BFIDA* and uses out-of-core PDBs for heuristic search.

Furthermore, the use of large PDBs in a group of PDBs is analyzed when maximizing over their h-values. Therefore parameters influencing the success of groups with different sized PDBs are analyzed. It is shown that the use of a large PDB is efficient if the available memory is sufficient to use it together with several smaller PDBs.

Inhaltsverzeichnis

1. Einleitung **9**

2. Grundlagen und Terminologie **13**

2.1. Graphensuche 13

2.2. Suchverfahren 15

2.2.1. Breitensuche 15

2.2.2. Tiefensuche 16

2.2.3. Iterative Tiefensuche 17

2.2.4. Heuristische Suchverfahren A* und IDA* 17

2.3. Frontier Search 19

2.4. Ausgewählte Heuristiken 20

2.4.1. Manhattan Distanz und Linear Conflicts 20

2.4.2. Pattern Datenbanken 21

2.4.3. Instance Dependent PDBs 24

2.4.4. Mehrere Heuristiken kombinieren 25

2.4.5. Symmetrien im Suchraum 26

2.5. Zusammenfassung 27

3. Massiv-parallele Suche mit MapReduce **29**

3.1. Forschungsstand 29

3.2. Das MapReduce-Paradigma 30

3.3. Breitensuche und das Erstellen von Pattern Datenbanken 32

3.3.1. Breadth-First Frontier Search mit MapReduce 32

3.3.2. Erstellen sehr großer PDBs 33

3.4. Heuristische Breitensuche mit MapReduce 35

3.4.1. Breadth-First Iterative-Deepening A* mit MapReduce 35

3.4.2. Suche mit großen PDBs 36

3.5. Das MR-Search Framework 38

3.6. Zusammenfassung 40

4. Effiziente Heuristiken – Eine empirische Analyse 41
4.1. Begrifflichkeiten und analytische Vorannahmen 42
4.2. Zentrale Ergebnisse 44
4.2.1. Einfluss der Gruppengröße 45
4.2.2. Einfluss der Gruppenzusammensetzung 47
4.3. Zusammenfassung 50

5. Anwendung der vorgestellten Algorithmen auf große Probleme 51
5.1. PDBs erstellen mit MR-BFFS 51
5.1.1. Eine vollständige 8+8+8 PDB 51
5.1.2. Zehn 6+6+6+6 PDBs 53
5.2. Heuristische Suche mit BFIDA* 54
5.2.1. BFIDA* vs. IDA* 54
5.2.2. Leistung der PDBs im Vergleich 56
5.2.3. Skalierbarkeit von MR-BFIDA* 58
5.2.4. MR-BFIDA* mit PDBs von Festplatte 63
5.3. Zusammenfassung 63

6. Fazit 67

A. Anhang 75
A.1. Methodischer Anhang 75
A.2. Zehn 6+6+6+6 Partitionierung für das 24er-Puzzle 78
A.3. 50 Probleminstanzen des 24er-Puzzles 80

1. Einleitung

Das Schiebepuzzle ist ein Rätsel, das im späten 19. Jahrhundert erfunden wurde. In der in Abbildung 1.1 dargestellten Variante besteht es aus 16 Feldern, wobei 15 mit nummerierten Spielsteinen belegt sind und ein Feld frei bleibt. Das Ziel des Spiels ist, die Steinchen in eine aufsteigende Reihenfolge zu bewegen. Die einzige Möglichkeit dies zu erreichen, ist das fortwährende Verschieben eines Steinchens auf die freie Position. Wenn das Puzzle von Hand gelöst werden soll, ist es schwierig, den Überblick über schon getätigte Züge oder sich wiederholende Anordnungen der Steinchen zu behalten. Das liegt an der Komplexität des Puzzles, denn obwohl es leicht zu verstehen ist und überschaubar erscheint, sind bei 15 Steinen circa 10^{13} Zustände möglich.

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Abbildung 1.1.: Das Schiebepuzzle mit 15 Spielsteinen

Aufgrund dieser Komplexität ist das Schiebepuzzle eine häufig verwendete Domäne für Untersuchungen in der Informatik und wird unter anderem für die Analyse von Suchalgorithmen verwendet. Suchen ist eine elementare Technik des Problemlösens. Um im Puzzle einen Pfad zur Lösung zu finden, können beispielsweise alle möglichen Züge probiert werden. Für schwere Probleme ist das Ausprobieren aller Möglichkeiten zu aufwendig. Pfade, die nicht zur Lösung führen, sollten daher möglichst schnell erkannt und von der Suche ausgeschlossen werden. Diese Aufgabe können Heuristiken übernehmen, indem die Länge des Pfades von einem Zustand zur Lösung geschätzt wird.

Da eine genaue Schätzung schwierig ist, wird vom Zustandsraum abstrahiert und ein kleinerer Raum gebildet. Dort können die Längen aller Pfade exakt bestimmt werden, die als untere Schranke für äquivalente Pfade im ursprünglichen Problem dienen.

Pattern Datenbanken (PDBs) verwenden diese Abstraktion des Zustandsraums. Je weniger das ursprünglichen Problem vereinfacht wird, desto genauer ist die gewonnene Abschätzung. Gleichzeitig wird die Berechnung der Wege in der Abstraktion komplexer und der benötigte Speicherplatz für die PDB größer. Mit bisherigen Mitteln gelingt es nicht, sehr große PDBs zu erschaffen, weil die Ressourcen eines Rechners dafür nicht ausreichen.

In der Informatik ist eine gängige Lösung für solche Ressourcenprobleme die Parallelisierung bzw. Verteilung der Aufgabe. So können Rechen- und Speicherkapazitäten zusammengefasst werden, damit das Erstellen der PDB dennoch gelingt. Um die Möglichkeiten der parallelen Datenverarbeitung effizient auszunutzen, sind spezielle Algorithmen nötig. Hier können Suchverfahren wie Tiefensuche oder Breitensuche als Grundlage dienen.

Eine Plattform zur massiv-parallelen Datenverarbeitung ist MapReduce, mit dem datenintensive Algorithmen einfach parallelisiert werden können. Diese Plattform nutzten Reinefeld und Schütt (2009), um einen Algorithmus zur massiv-parallelen Breitensuche in beliebigen Domänen zu entwickeln.

Ziel der vorliegenden Arbeit ist nun die Erweiterung dieses Algorithmus für das Erstellen von PDBs. Der Algorithmus soll dabei ebenfalls beliebig skalieren. Des Weiteren soll eine Methode zur Verwendung sehr großer PDBs in der Suche entwickelt werden. Hierbei werden PDBs genutzt, die größer sind als der verfügbare Hauptspeicher. Um die Leistungsfähigkeit der Algorithmen zu zeigen, sollen sehr große PDBs erzeugt und mit diesen schwere Probleme gelöst werden.

Neben der parallelen Suche und dem Erzeugen von PDBs beschäftigt sich die Arbeit mit effizienten Heuristiken. Besonders wirkungsvoll sind Heuristiken, wenn mehrere gleichzeitig verwendet werden. Da PDBs aber Speicherplatz benötigen, wird untersucht, welche PDBs zur Anwendung kommen können, um eine möglichst gute Leistung zu erhalten. Dazu wird in dieser Arbeit ein Effizienzbegriff definiert und eine umfassende Untersuchung in einer kleinen Domäne durchgeführt. Es soll dabei analysiert werden, wie sich Gruppen verschieden großer PDBs bei der Suche verhalten und welche Faktoren einen Einfluss auf die Leistung haben.

Struktur der Arbeit

Ausgangspunkt des folgenden Kapitels ist die Erläuterung grundlegender Begriffe zum Thema Suche und die Vorstellung von Techniken zur Problemmodellierung. Anschlie-

ßend werden Suchverfahren eingeführt, auf denen die später vorgestellten Algorithmen basieren. Ein wichtiger Aspekt bei der Suche ist das Einsparen von Speicherplatz. Um Suchverfahren diesbezüglich zu verbessern, kann die im Anschluss vorgestellte Suchtechnik Frontier Search hilfreich sein. Da die heuristische Suche im Mittelpunkt der Arbeit steht, endet das zweite Kapitel mit der Vorstellung einiger Heuristiken.

Das dritte Kapitel gibt zunächst einen Überblick über den Forschungstand zum Thema parallele Suche und führt dann in das MapReduce-Paradigma ein. Dieses ist die Grundlage für den anschließend vorgestellten Algorithmus zur Breitensuche, mit dem sehr große PDBs erstellt werden können. Danach wird ein Algorithmus zur heuristischen Breitensuche eingeführt, der ebenfalls auf MapReduce basiert. Daraufhin wird das MR-Search Framework vorgestellt, das im Laufe der vorliegenden Arbeit entwickelt wurde und die eingeführten Algorithmen mit MapReduce implementiert.

Im vierten Kapitel geht es um die Definition des Effizienzbegriffs, der Grundlage für die anschließend vorgestellten Versuche im 8er-Puzzle ist.

Im Anschluss daran werden die vorgestellten Algorithmen auf große Probleme angewandt. Dies geschieht durch die Dokumentation der Erzeugung mehrerer PDBs. Untersuchungen bezüglich der heuristischen Suche und einiger Eigenschaften des entwickelten Algorithmus schließen dieses fünfte Kapitel ab.

Der letzte Teil der Arbeit fasst die Ergebnisse zusammen und gibt einen Ausblick auf zukünftige Forschungsfelder.

2. Grundlagen und Terminologie

In diesem Kapitel werden wichtige Grundlagen und Begriffe der Suche eingeführt. Dazu wird zunächst eine geeignete Problemmodellierung vorgestellt, auf deren Basis eine Graphensuche möglich wird. Im Anschluss daran wird auf die grundlegenden Suchverfahren Breitensuche, Tiefensuche und iterative Tiefensuche sowie auf die heuristischen Suchverfahren A* und IDA* eingegangen. Der darauf folgende Abschnitt beschreibt die Frontier Search, welche eine Suchtechnik zur speichereffizienten Breiten- und Bestensuche ist. Der vierte Teil führt einige Heuristiken ein, die für die zielgerichtete Suche eine große Rolle spielen.

2.1. Graphensuche

Um ein Problem lösen zu können, muss es erfasst und mit geeigneten Methoden modelliert werden. Simon und Newell (1972) haben sich mit der maschinellen Lösung von Problemen auseinandergesetzt, die zu komplex für die kognitiven Fähigkeiten des Menschens sind. In diesem Zusammenhang führten sie den Begriff des Problemraums ein, der ein Mittel zur Modellierung ist. Demnach besteht ein *Problemraum* (oder *Zustandsraum*) aus einer Menge von Zuständen und einer Menge von partiellen Funktionen (*Operatoren*), die von der Menge der Zustände in die Menge der Zustände abbilden.

Im Anschluss werden einige weitere Begriffsdefinitionen vorgenommen.

- Ein Zustand z_2 ist ein *Nachfolger* von einem Zustand z_1, wenn es einen Operator *op* gibt, so dass gilt: $op(z_1) = z_2$. Der Zustand z_1 entspricht wiederum einem *Vorgänger* von z_2. Ein Zustand kann dabei mehrere Nachfolger haben.

- Ein Operator *op* ist genau dann *umkehrbar*, wenn es einen weiteren Operator op' gibt, so dass gilt: $op'(op(z1)) = z1$. In dieser Arbeit werden alle Operatoren als umkehrbar angenommen.

- Ein Problemraum mit einem ausgezeichnetem Startzustand und einer Menge Zielzustände ist ein *Problem*. Eine Sequenz von Operatoren, die den Startzustand in

einen Zielzustand überführt, *löst* das Problem.

- Die Suche mit vorgegebenem Ziel wird in dieser Arbeit als *zielgerichtete Suche* bezeichnet.
- Der zuvor definierte Zustandsraum spannt einen impliziten ungerichteten Graphen auf, den *Zustandsgraphen*. Zur Lösung eines Problems muss dieser Graph nach Zielzuständen durchsucht werden. Der Zustandsgraph ist implizit, weil er durch die Menge der Zustände und die Menge der Operatoren und nicht explizit mittels einer Adjazenzliste oder -matrix beschrieben wird.
- Zwei Knoten v_1 und v_2 des Zustandsgraphen sind genau dann durch eine Kante verbunden, wenn v_2 ein Nachfolger oder Vorgänger von v_1 ist.
- Im Laufe der Suche wird ein *Suchgraph* erzeugt, welcher ein Teilgraph des Zustandsgraphen ist. Der Suchgraph enthält zum einen alle bereits besuchten Knoten. Zum anderen enthält er eine Kante zwischen zwei Knoten v_1 und v_2, wenn die Expansion von v_1 zur Erzeugung von v_2 führt oder andersherum.
- Einen Knoten zu *expandieren* heißt, seine Nachfolger zu erzeugen, während einen Knoten *erzeugen* bedeutet, die konkrete Datenstruktur für diesen Knoten anzulegen.
- Die *Tiefe* eines Knotens v in einem Graphen mit ausgezeichnetem Startknoten s entspricht der Länge des kürzesten Pfades von s nach v.

Das *Schiebepuzzle* ist ein Permutationsproblem. In der Kombinatorik versteht man unter einer n-stelligen Permutation die Anordnung einer Menge durch Vertauschen ihrer n Elemente. Jeder Operator eines Permutationsproblems vertauscht die Positionen mehrerer Elemente. In der Regel sind jedoch nicht alle Vertauschungen erlaubt. Die jeweilige Domäne gibt konkrete Beschränkungen für gültige Operationen vor.

Das n-Puzzle besteht aus n durchnummerierten Spielsteinen, die in einem Quadrat angeordnet sind, wobei ein Feld frei gelassen wird (das *Blank*). Das Ziel des Spiel ist, den kürzesten Pfad zu finden, der die Steinchen in eine aufsteigende Reihenfolge bringt. Abbildung 2.1 zeigt die Lösung des 15er-Puzzles[1] und eine Probleminstanz. Eine gültige Operation ist hierbei, ein horizontal oder vertikal an das Blank angrenzendes Steinchen auf die Position des Blanks zu verschieben. Die Verwendung des Puzzles für die Analyse

[1] Zur besseren Lesbarkeit wird in dieser Arbeit z.B. das 15-Puzzle als 15er-Puzzle bezeichnet.

	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

	1	2	3
4	5	6	7
9	15	11	8
10	14	13	12

Abbildung 2.1.: Lösung (links) und Zufallsinstanz (rechts) des 15er-Puzzles

von Suchverfahren und Heuristiken bietet sich an, weil es trotz der einfachen Regeln zur Beschreibung des Problems einen sehr großen Zustandsraum aufspannt. Mit zunehmender Teilchenzahl nimmt außerdem die Komplexität stark zu. Während das 8er-Puzzle mit 181440 Zuständen noch relativ überschaubar ist, hat das 15er-Puzzle bereits 10^{13} Zustände. Das 24er-Puzzle ist mit 10^{25} Zuständen wesentlich schwieriger. Diese Arbeit greift an einigen Stellen beispielhaft und für die Evaluation der vorgestellten Algorithmen grundlegend auf das 8er- und 24er-Puzzle zurück.

2.2. Suchverfahren

Im Folgenden werden verschiedene Algorithmen vorgestellt, die zum Durchsuchen von Graphen genutzt werden können. Für diese Arbeit spielt die Breitensuche eine zentrale Rolle, da die im dritten Kapitel vorgestellten Algorithmen auf diesem Suchverfahren basieren.

2.2.1. Breitensuche

Die *Breitensuche* ist ein Suchverfahren, das sich besonders für das komplette Durchsuchen eines Problemraums eignet, da jeder Knoten genau einmal besucht wird. Bei der Breitensuche wird ein Knoten der Tiefe d erst dann expandiert, wenn alle Knoten der Tiefe $d-1$ expandiert wurden. Dieses Vorgehen entspricht der Abarbeitung einer *First In – First Out* Warteschlange (*FIFO-Queue*). Zu Beginn der Suche wird der Startzustand in die FIFO-Queue gestellt. In jedem Schritt expandiert die Breitensuche den vordersten Knoten der Queue und stellt die Nachfolger des Knotens an deren Ende.

Das erneute Expandieren bereits besuchter Knoten wird in der Breitensuche durch zwei Listen vermieden, die als OPEN- bzw. CLOSED-Liste bezeichnet werden.

In der ersten werden Knoten gespeichert, die erzeugt, jedoch noch nicht expandiert wurden. Letztere beinhaltet alle bereits vollständig expandierten Knoten. Ein Nachfolger darf nur dann der OPEN-Liste hinzugefügt werden, wenn er nicht schon Teil der OPEN- oder CLOSED-Liste ist, ergo in einem früheren Durchlauf erzeugt wurde. Auf diese Weise erkennt die Breitensuche Zyklen im Zustandsgraphen.

Der Speicherbedarf eines Suchverfahrens zum Lösen eines Problems ist durch zwei Faktoren der Problemkomplexität bestimmt: Zum Einen gibt der Verzweigungsfaktor b die durchschnittliche Anzahl an Nachfolgern an, die durch die Menge der Operatoren erzeugt werden. Zum Anderen ist die Länge des kürzesten Pfades d von der Probleminstanz zu einer Lösung entscheidend. Da bei der Breitensuche alle bisher expandierten Knoten gespeichert werden müssen, um Zyklen im Suchgraphen zu vermeiden, entspricht der asymptotische Speicherbedarf der Breitensuche $\mathcal{O}(b^d)$. Der benötigte Speicher wächst exponentiell mit der Tiefe der Lösung.

Wenn eine Menge von Zielknoten vorgegeben ist, also nicht der gesamte Suchraum erkundet werden soll, sind die beiden Kriterien *Vollständigkeit* und *Optimalität* wichtig. Ersteres gibt an, ob eine Lösung gefunden wird, wenn eine solche vorhanden ist. Optimalität bedeutet, dass keine Lösung mit geringerer Tiefe existiert als die Gefundene. Die Breitensuche erfüllt beide Kriterien.

2.2.2. Tiefensuche

Während die Breitensuche in den Zustandsraum schrittweise tiefer dringt, durchsucht die *Tiefensuche* zunächst Knoten mit großer Tiefe. Dieses Suchverfahren erzeugt solange die Nachfolger des jeweils zuletzt erzeugten Knotens, bis keine weiteren Nachfolger mehr existieren. Dann erfolgt ein Backtracking bis zur letzten nicht vollständig expandierten Tiefe. Dieses Vorgehen entspricht der Abarbeitung einer *Last In – First Out* Warteschlange (*LIFO-Queue* oder *Stack*). Hierfür wird keine OPEN- oder CLOSE-Liste benötigt, da die Tiefensuche immer nur einen Pfad vom Startknoten aus untersucht und die zuletzt erzeugten Knoten auf dem Stack speichert. Dieser kann sowohl eine separate Datenstruktur als auch, in der rekursiven Variante der Tiefensuche, der Programmstack sein. Der Speicherbedarf der Tiefensuche ist sehr gering, denn es müssen lediglich die Knoten auf dem Pfad von der Wurzel bis zum aktuellen Knoten gespeichert werden. Die Speicherkomplexität beträgt somit $\mathcal{O}(d)$.

Ein Nachteil der Tiefensuche ist, dass diese weder vollständig ist, noch zwangsläufig eine optimale Lösung findet. Außerdem kann die Tiefensuche durch Zyklen im Zustands-

graphen nicht terminieren. Um das zu verhindern, kann die Suche auf eine maximale Tiefe begrenzt werden, bis zu der weitere Nachfolger expandiert werden.

2.2.3. Iterative Tiefensuche

Eine solche Beschränkung der Tiefe nimmt die iterative Tiefensuche vor, deren allgemeine Anwendbarkeit Korf (1985*a*) aufgezeigt hat. Es handelt sich um eine Kombination aus Breiten- und Tiefensuche, wobei in jedem Schritt eine begrenzte Tiefensuche bis zu einem Schwellwert (*Threshold*) der Tiefe d durchgeführt wird. Der Threshold wird solange erhöht, bis die begrenzte Tiefensuche eine Lösung findet. Durch das sukzessive Erhöhen des Thresholds ist die iterative Tiefensuche vollständig und die gefundene Lösung optimal. Da in jedem Schritt eine einfache Tiefensuche durchgeführt wird, ist der Speicherbedarf bei diesem Verfahren ebenfalls $\mathcal{O}(d)$.

Die iterative Tiefensuche verbindet damit die Vorteile der Breiten- und der Tiefensuche. Demzufolge hat sie eine geringe Speicherkomplexität und ist dennoch vollständig und optimal. Des Weiteren werden, im Gegensatz zur einfachen Tiefensuche, Zyklen korrekt behandelt.

Korf hat gezeigt, dass kein anderes Suchverfahren weniger Knoten expandiert oder weniger Zeit benötigt, wenn die zu durchsuchende Datenstruktur ein Baum ist. In diesem Spezialfall ist das Erkennen von Duplikaten nicht nötig, da ein Baum per Definition keine Zyklen enthält. In Datenstrukturen mit Zyklen können Duplikate im Suchgraph einen wesentlichen Overhead für die Suche darstellen. So konnten Reinefeld und Schütt (2009) mit einem vollständigen Durchsuchen des 15er-Puzzles nachweisen, dass der Zustandsgraph dieser Domäne zu circa einem Drittel aus Duplikaten besteht. In diesem Fall ist das Erkennen von Duplikaten sehr wichtig, um unnötigen Aufwand bei der Suche zu vermeiden.

2.2.4. Heuristische Suchverfahren A* und IDA*

Die bisher vorgestellten Suchverfahren bewerten alle Nachfolger eines Knotens gleich. Ausschließlich die Tiefe steht als Bewertungsgrundlage zur Verfügung. Diese Art der Suche heißt blinde oder *uninformierte Suche* und hat den Nachteil, dass auch nicht optimale Pfade erkundet werden, zum Beispiel auch solche, die auf gar kein Ziel zulaufen, sondern sich davon entfernen. Genau an diesem Punkt kann die zielgerichtete Suche mit Hilfe einer Kostenschätzfunktion (*Heuristik*) diese Pfade teilweise auszuschließen. Wie viele irrelevante Pfade verworfen werden können, hängt von der Qualität der Heuristik

ab. Die Kombination aus zielgerichteter Suche und Heuristik wird als *informierte* oder *heuristische* Suche bezeichnet.

Eine Heuristik schätzt die benötigten Kosten $h(n)$, um von dem betrachteten Knoten n zu einem Zielknoten zu gelangen. Die bereits benötigten Kosten vom Startknoten zu n werden mit $g(n)$ angegeben. Zusammen ergeben sich die Gesamtkosten $f(n)$ eines Knotens als $f(n) = g(n) + h(n)$.

Für die heuristische Suche sind die folgenden Eigenschaften von Heuristiken von Bedeutung:

- Eine Heuristik ist *zulässig* (engl. admissible), wenn sie die exakten Kosten h^* nie überschätzt: $\forall n\, h(n) \leq h^*(n)$ (vgl. Pearl, 1984, S. 77).
- Die exakten Kosten h^* werden auch als *perfekte Heuristik* bezeichnet.
- Neben der Eigenschaft der Zulässigkeit existiert das Merkmal der Monotonie (oder Konsistenz). Eine Heuristik ist *monoton*, wenn für jeden Nachfolger n' von n gilt: $h(n) \leq g(n, n') + h(n')$. Die Kosten von n nach n' notiert hier $g(n, n')$ (vgl. Pearl, 1984, S. 82f).

Bestensuchverfahren (kurz: *Bestensuche*) expandieren in jedem Schritt den Knoten mit den kleinsten Gesamtkosten. Eine Form der Bestensuche ist der Algorithmus A* von Hart et al. (1968), der eine heuristische Variante der Tiefensuche darstellt und zur Bestimmung eines kürzesten Pfades zwischen zwei Zuständen genutzt wird. A* expandiert nicht, wie bei der Tiefensuche üblich, den zuletzt erzeugten Nachfolger, sondern den mit den geringsten Gesamtkosten $f(n)$ und findet eine optimale Lösung, wenn die Heuristik zulässig ist.

Dazu verwendet A*, wie die Breitensuche, eine OPEN- und eine CLOSED-Liste. In diesen Listen wird ein Knoten n zusammen mit $f(n)$ und einen Zeiger auf seinen Vorgänger gespeichert. Es ist möglich, dass n auf zwei verschiedenen Pfaden der Länge g bzw. g' erreichbar ist. Wird ein Pfad der Länge g zu n entdeckt und ist n bereits in der OPEN- oder CLOSED-Liste mit der Tiefe g', müssen beide Tiefen verglichen werden. Wenn $g \geq g'$, kann der neu gefundene Pfad verworfen werden, weil bereits ein kürzerer bzw. gleich langer Pfad existiert. Wenn die Heuristik nicht monoton ist, kann $g < g'$ sein. Dann müssen die Kosten $f(n)$ und der Zeiger auf den Vorgänger von n aktualisiert werden. Befindet sich n bereits in der CLOSED-Liste, muss er in die OPEN-Liste verschoben werden.

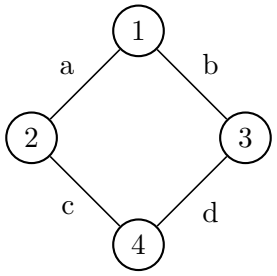


Abbildung 2.2.: Beispiel eines Zustandsgraphen

Eine iterative Variante von A* hat Korf (1985*a*) als Erster implementiert und *iterative-deepening A** (IDA*) genannt. Initial wird hierbei der Threshold[2] auf den Heuristikwert des Startzustandes als unterste Schranke für die Länge des kürzesten Lösungspfads gesetzt. Ein Knoten wird nur dann expandiert, wenn $g(n) + h(n) < threshold$.

2.3. Frontier Search

Bei der Frontier Search handelt es sich nicht um ein eigenständiges Suchverfahren, sondern vielmehr um eine Suchtechnik. Sie ist auf die Bestensuche und die einfache Breitensuche anwendbar. Der vorangegangene Abschnitt hat unter anderem herausgestellt, dass diese Suchverfahren eine Datenstruktur zur Speicherung *aller* bereits besuchten Knoten nutzen. Bei großen Problemräumen kann diese Datenstruktur sehr umfangreich sein, weil nicht nur die Knoten der aktuellen Tiefe g (*äußerer Bereich*) gespeichert werden, sondern auch die Knoten aller Tiefen kleiner als g (*innerer Bereich*).

Der benötigte Speicherplatz kann durch die *Frontier Search*, die erstmals von Korf (1999) vorgeschlagen wurde, verringert werden. Statt der sonst üblichen OPEN- und CLOSED-Liste wird bei dieser Technik lediglich die OPEN-Liste gespeichert. Die CLOSED-Liste kann ersetzt werden, indem an jedem Knoten die Menge der Operatoren gespeichert wird, die bereits zu diesem Knoten führten. Die Frontier Search erzeugt nur diejenigen Nachfolger, für die bereits genutzte Operatoren nicht rückgängig gemacht werden. Jeder ungenutzte Operator führt zu einem Knoten der nächsten Tiefe.

Das folgende Beispiel verdeutlicht das Vorgehen, in dem die Frontier Search als Breitensuche durchgeführt wird. Dieses Verfahren heißt *Breadth-First Frontier Search* (BFFS). Abbildung 2.2 zeigt einen Zustandsgraphen mit den Zuständen 1, 2, 3 und 4 sowie den umkehrbaren Operatoren a, b, c und d. Zustand 1 soll der Startzustand sein, die Menge

[2] Vgl. Kapitel 2.2.3.

der Zielzustände sei $\{4\}$. Wird auf diesem Graphen eine Frontier Search durchgeführt, beginnt der Algorithmus zunächst mit dem Tupel des Startknotens und der leeren Menge (ungenutzter Operatoren) $(1, \emptyset)$. Anschließend werden die Nachfolger von 1 bestimmt und mit dem jeweiligen Operator (a bzw. b) markiert. Es entstehen: $(2, \{a\})$ und $(3, \{b\})$. Die Ebene der Tiefe eins ist nun vollständig erkundet. Betrachtet die Suche im Anschluss beispielsweise Knoten 2, sind dessen Nachfolger 1 und 4. Für ersteren muss allerdings der Operator a rückgängig gemacht werden, so dass einzig Zustand 4 zu erzeugen bleibt. Der Zielzustand wird erreicht.

Bei der Frontier Search wird nur der äußere Bereich der Suche, die *Suchfront*, gespeichert. Das Wissen, ob ein Knoten bereits besucht wurde und sich im Inneren des Suchbereichs befindet, ist in der Menge der Operatoren des jeweiligen Knotens kodiert. Folglich können alle Knoten parallel expandiert werden. Allerdings gehen die Informationen verloren, wie die Suche zu einem Knoten gelangt ist. Wenn das Ziel der Suche das Finden des kürzesten Pfades zwischen zwei Knoten ist, stellt sich folgendes Problem: Am Ende der Suche ist zunächst nur die Länge des kürzesten Pfades bekannt. Der Pfad selbst kann wiederhergestellt werden, indem ein Zustand z_p entlang des kürzesten Pfades gespeichert wird. Im Anschluss der Suche kann die Frontier Search rekursiv vom Startzustand zu z_p und von diesem zum Zielzustand ausgeführt werden. Die vorliegende Arbeit beschränkt sich allerdings auf die Länge des kürzesten Pfades in ungerichteten Graphen mit gleichförmigen Kantenkosten, weshalb für eine ausführliche Diskussion zur Wiederherstellung des kürzesten Pfades auf Korf et al. (2005, S. 722 ff.) verwiesen wird.

2.4. Ausgewählte Heuristiken

Der folgende Abschnitt stellt einige Heuristiken vor, die unter anderem für die Suche im Schiebepuzzle verwendet werden können. Ferner werden die Möglichkeit zur Kombination mehrerer Heuristiken sowie das Ausnutzen von Symmetrien im Suchraum diskutiert.

2.4.1. Manhattan Distanz und Linear Conflicts

Eine einfache Heuristik stellt die *Manhattan Distanz* dar, welche erstmals durch Johnson und Story (1879) Erwähnung fand. Sie kann unter anderem auf das Schiebepuzzle angewendet werden. In dieser Domäne sind die Beschränkungen für einen gültigen Zug eines Teilchens von der Position x zur Position y, dass zum einen x und y benachbart sind und zum anderen die Position y frei ist. Zur Berechnung der Manhattan Distanz wird die

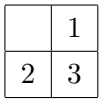


Abbildung 2.3.: Position im 3er-Puzzle

zweite Beschränkung aufgehoben. Jedes Teilchen kann auf jede beliebige benachbarte Position bewegt werden. Die Kostenfunktion, um jedes Teilchen auf seine Position im Zielzustand zu bewegen, ist leicht zu berechnen und liefert eine grobe Abschätzung der Entfernung einzelner Zustände zum Ziel. (vgl. Felner et al., 2004, S. 281).

Eine von Hansson et al. (1992) vorgeschlagene Erweiterung der Manhattan Distanz sind lineare Konflikte (*Linear Conflicts*). Abbildung 2.1 (S. 15) zeigt einen linearen Konflikt im Schiebepuzzle. Die Teilchen 13 und 14 befinden sich beide in der richtigen Zeile, jedoch in umgekehrter Reihenfolge. Die Manhattan Distanz für diese beiden Teilchen beträgt 2. Um die beiden jedoch aneinander vorbeizuführen, muss ein Teilchen zwei zusätzliche Züge machen, so dass der Heuristikwert der Teilchen auf 4 erhöht werden kann.

Im Allgemeinen stellt eine Heuristik eine exakte Kostenfunktion für ein vereinfachtes Modell des ursprünglichen Problems dar (vgl. Pearl, 1984, S. 113ff). Dabei gilt, je weniger das Modell das Problem vereinfacht, desto stärker ist die resultierende Schätzfunktion. Der erhöhte Heuristikwert für die Kombination von Manhattan Distanz und Linear Conflicts wurde durch Hinzufügen einer weiteren Beschränkung zum vereinfachten Modell der Manhattan Distanz erreicht.

2.4.2. Pattern Datenbanken

Pattern Datenbanken (PDBs) sind leistungsfähige Heuristiken. Sie werden hier am Beispiel des Schiebepuzzles eingeführt[3]. PDBs basieren auf einer Abstraktion des Problems, bei der nur einige Spielsteine berücksichtigt und die anderen durch Platzhaltersteine ersetzt werden. Es gelten folgende grundlegende Definitionen:

- Angelehnt an Korf (1985*b*) ist ein Zustand im n-Puzzle über einer Menge von Zustandsvariablen $\{v_0, ..., v_n\}$ mit dem Wertebereich $D = \{1, ..., n\} \cup \{blank\}$ definiert. Die v_i entspricht der i-ten Stelle einer Position des Puzzles. Ein Zustand ist ein Vektor von Zustandsvariablen $(v_0, ..., v_n) \in D^{n+1}$. Die Position des 3er-Puzzles aus Abbildung 2.3 wird in dieser Schreibweise mit $s = (blank, 1, 2, 3)$ angegeben.

[3]Für eine allgemeine, formale Definition sei auf Yang et al. (2008) verwiesen.

- Für eine PDB wird eine Teilmenge $P \subseteq D$ der Spielsteine berücksichtigt. Alle anderen werden durch Platzhaltersteinchen X ersetzt. Die Projektion einer Stelle v_i $\sigma_P : D \to P \cup \{X\}$ ist definiert als:

$$\sigma_P(v_i) = \begin{cases} blank, & \text{wenn } v_i = blank \\ v_i, & \text{wenn } v_i \in P \\ X, & \text{sonst} \end{cases}$$

  Die Projektion $\sigma_P : D^{n+1} \to (P \cup \{X\})^{n+1}$ eines Zustands $s = (v_0, ..., v_n)$ ist definiert als $\sigma_P(s) = (\sigma_P(v_0), ..., \sigma_P(v_n))$.

  Die Projektion des o. g. Zustands s im 3er-Puzzle auf $P = \{1\}$ lautet $\sigma_P(s) = (blank, 1, X, X)$. $\sigma_P(s)$ ist ein *Pattern* des Zustands s. Bei einem *k-Pattern* gilt $|P| = k$. Die teilweise Spezifikation des Zielzustands heißt *Zielpattern*. Im Schiebepuzzle unterscheiden sich zwei Pattern durch die Lage ihrer Patternsteine und der des Blanks. Es wird hier als zusätzliches nullter Patternstein angenommen.

- Die durch das Pattern berücksichtigten Spielsteine $p \in P$ heißen *Patternsteine*. Da die Steine, die nicht zum Pattern gehören, nicht mehr unterscheidbar sind, werden mehrere Zustände auf das gleiche Pattern abgebildet.

- Der *Patternraum* entsteht implizit aus dem Zustandsraum durch Anwenden der Projektion σ_P auf jeden Zustand. Die Menge der Nachfolgerfunktionen wird dabei wie folgt definiert. Ein Pattern p_1 ist genau dann der Nachfolger von einem Pattern p_2, wenn zwei Zustände z_1, z_2 und ein Operator *op* im ursprünglichen Problem existieren, so dass $\sigma_P(z_1) = p_1$, $\sigma_P(z_2) = p_2$ und $op(z_2) = z_1$.

- Der Patternraum eines k-Patterns des n-Puzzles hat $\frac{n!}{(n-k)!}$ Zustände.

Eine PDB ist eine Lookup-Tabelle, die in Bezug auf ein Zielpattern definiert wird. Sie wird durch eine Breitensuche des Patternraums ausgehend vom Zielpattern erzeugt. Die PDB beinhaltet für jedes Pattern die exakten Kosten, die benötigt werden, um das Pattern in das Zielpattern zu überführen. Da das Pattern eine Abstraktion des ursprünglichen Problems ist, ergibt sich eine zulässige Heuristik für das ursprüngliche Problem.

In der einfachen Variante einer PDB nach Culberson und Schaeffer (1998) werden die Kosten aller Operatoren unabhängig davon gezählt, ob sie Pattern- oder Platzhaltersteine bewegen. Die Heuristikwerte zweier PDBs dürfen jedoch nicht addiert werden,

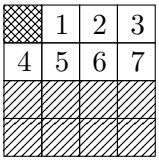


Abbildung 2.4.: 7-Pattern im 15er-Puzzle

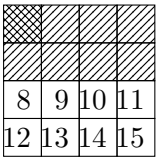


Abbildung 2.5.: 8-Pattern im 15er-Puzzle

selbst wenn ihre Zielpattern disjunkt sind. Denn neben den Bewegungen der Patternsteinen, fließen zusätzlich die Bewegungen der Platzhaltersteine in den Heuristikwert ein. In der Summe dieser Heuristikwerte werden Bewegungen von Platzhaltersteine, die in der anderen PDB Patternsteine sind, doppelt gezählt. Die Heuristik ist nicht mehr zulässig.

Durch eine Anpassung der Kostendefinition erweiterten Korf und Felner (2002) die bisher beschriebenen PDBs zu *additiven* PDBs. Hierbei werden nicht die Kosten aller Operatoren summiert, sondern ausschließlich die Kosten der Operatoren, die eine Patternsteine bewegen. Zwei PDBs sind somit additiv, d.h. ihre Heuristikwerte können addiert werden, wenn ihre Zielpattern disjunkt sind und die Kostenfunktion nur Veränderungen der Patternsteine widerspiegelt.

Abbildung 2.4 zeigt exemplarisch ein Pattern im 15er-Puzzle. Es ist über die Teilchen 1 bis 7 definiert. Alle anderen Teilchen sind Platzhaltersteinchen. Ein weiteres Pattern könnte, wie in Abb. 2.5 zu sehen, über die Teilchen 8 bis 15 definiert sein. Da beide Pattern disjunkt sind, können – mit entsprechender Kostenfunktion – ihre Heuristikwerte addiert werden. Die zusammengefasste additive PDB wird als 7+8 PDB bezeichnet.

Für die Repräsentation von PDBs im Hauptspeicher beschreiben Felner et al. (2007) zwei Varianten der Speicherung für Permutationsprobleme. In der ersten Variante wird die PDB eines k-Patterns in einem k-dimensionalen Array gespeichert. Die Position des i-ten Patternsteins dient als i-ter Index in das Array. So wäre der Heuristikwert eines Patterns $(X, 1, 0, X)$ im Array unter h[2][1] gespeichert. Diese Methode zeichnet sich durch schnellen Zugriff aus. Ein Nachteil ist hingegen, dass die PDB mit n^k Einträgen größer als der Patternraum ist, da Einträge mit zwei oder mehreren gleichen Indizes keine gültige Permutation widerspiegeln. Diese Variante wird *sparse-Mapping* genannt. Die zweite Variante, das *compact-Mapping*, speichert die PDB in einem eindimensionalen Array, das exakt so groß ist wie der Patternraum. Der Index wird nach einem Algorithmus von Korf und Schultze (2005) berechnet. Als grundlegende Idee soll das Pattern zu dem Index der lexikographischen Anordnung aller Pattern abgebildet werden. Vorteil dieses Vorgehens ist, dass es keine ungenutzten Indizes gibt. Die Berechnung des Index

ist allerdings rechenintensiv, so dass der Zugriff in die PDB verlangsamt wird (vgl. Felner et al., 2007, S. 220f).

Je mehr Steine des Puzzles eine PDB umfasst, desto näher sind ihre Einträge an der optimalen Lösung und desto leistungsfähiger ist die Heuristik (vgl. Holte und Hernádvölgyi, 1999; Korf, 1997). Die größtmögliche PDB ist eine Datenbank, die alle n Steine beinhaltet und somit die perfekte Heuristik h^* bildet. Das exakte Gegenteil stellen n PDBs mit je einem Element dar. Dies gleicht der Manhattan Distanz.

Ein Vorteil von PDBs ist ihre Wiederverwendbarkeit. Wenn sie einmal erstellt wurden, können sie für jede Probleminstanz einer Domäne verwendet werden, solange das Zielpattern gleich bleibt. Der relative Aufwand zur Erstellung der PDB sinkt mit jeder gelösten Instanz und kann schließlich vernachlässigt werden. Ein Nachteil ist die Größe von PDBs. Eine vollständige 8+8+8 PDB für das 24er-Puzzle benötigt bei compact-Mapping beispielsweise 2071 GByte Hauptspeicher.

In ihrer Arbeit beschreiben Felner et al. (2007) mehrere Möglichkeiten, PDBs zu komprimieren, um Platz zu sparen. Der Grundgedanke ist, ähnliche Pattern zusammenzufassen und das Minimum ihrer Heuristikwerte zu speichern. Im Schiebepuzzle ist beispielsweise eine Komprimierung nach der Position des Blanks möglich. Wie oben bereits ausgeführt, unterscheiden sich dort Pattern durch die Lage ihrer Patternsteine und der des Blanks. Wird die Position des Blanks jedoch nicht berücksichtigt, fallen mehrere Pattern zusammen. Da somit die Zahl der Patternsteine verringert wird, sinkt durch diese Komprimierung der Speicherbedarf der o. g. 8+8+8 PDB auf 120 GByte. Diese Art der Komprimierung wird durchgehend in der vorliegenden Arbeit verwendet.

2.4.3. Instance Dependent PDBs

Für das Lösen eines individuellen Problems ist nur ein Bruchteil der gespeicherten PDB nötig. Diesen Umstand machen sich Felner und Adler (2005) mit *Instance Dependent PDBs* (IDPDBs) zu Nutze. Sie verwenden keine im Voraus berechnete PDB, sondern erzeugen für jede Probleminstanz eine individuelle IDPDB. Die Suche teilt sich in zwei Phasen. In der ersten Phase wird die IDPDB im Patternraum aufgebaut, indem eine heuristische Suche vom Zielpattern zum Pattern der Probleminstanz erfolgt (Rückwärtssuche). Für jedes in der Suche auftretende Pattern wird, analog zur vollständigen PDB, die geringste Tiefe in der IDPDB gespeichert, mit der das Pattern im Suchgraphen aufgetreten ist. Diese Phase wird auch *Sekundärsuche* genannt. Im Anschluss findet die *primäre Suche* von der Probleminstanz zum Zielzustand statt (Vorwärtssuche), wobei

auf die Heuristikwerte der IDPDB zugegriffen wird. Während der Suche kann es vorkommen, dass für ein Pattern, das zur Bestimmung des Heuristikwerts notwendig ist, kein Eintrag in der IDPDB vorhanden ist. In diesem Fall kann eine andere Heuristik verwendet werden. Der Erfolg der IDPDB hängt von der Trefferquote ab, die angibt, in wieviel Prozent der Fälle ein Pattern erfolgreich in der IDPDB nachgeschlagen werden konnte. Diese Trefferquote kann erhöht werden, indem nach der ersten Sekundärsuche weitere Suchdurchläufe mit erhöhtem Threshold durchgeführt werden. Obwohl das Pattern der Probleminstanz mit einem gegebenen Threshold gefunden werden konnte, besteht die Möglichkeit, diesen weiter zu erhöhen. Das steigert die Zahl der besuchten Knoten je Tiefe und somit der Pattern in der IDPDB. Das Ziel der Arbeit von Felner und Adler war es, eine IDPDB an den verfügbaren Hauptspeicher anzupassen. Sie erweitern die IDPDB solange, bis der komplette verfügbare Hauptspeicher ausgenutzt ist.

Im Gegensatz zur (vollständigen) PDB kann bei einer IDPDB die Zeit zum Erstellen der Datenbank nicht vernachlässigt werden, weil sie pro Probleminstanz erneut berechnet werden muss. Um den Erfolg von IDPDBs zu bewerten, sind daher die Anzahl der Knoten, die für die Sekundärsuche expandiert wurden, zu der Zahl der Knoten hinzuzuaddieren, die in der Primärsuche expandiert wurden. Eine weitere Möglichkeit besteht darin, die gesamte benötigte Zeit für die Suche zu vergleichen.

2.4.4. Mehrere Heuristiken kombinieren

Eine Heuristik h schätzt die Distanz eines Zustands z zur Lösung. Die Genauigkeit der Schätzung ist für verschiedene Zustände unterschiedlich gut, wobei $0 \leq h(z) \leq h^*(z)$. Ein Knoten wird bei IDA* nur dann erzeugt, wenn $g(n) + h(n) < threshold$ (vgl. Kapitel 2.2.4). Je näher also ein Heuristikwert für einen Zustand an dem Wert der perfekten Heuristik für diesen Zustand ist, desto größer ist die Wahrscheinlichkeit, dass dieser Wert zum Abschneiden eines Teilbaums führt. Durch Kombination mehrerer Heuristiken ist es möglich, die Zahl der besuchten Knoten zu verringern. So können zum Beispiel die Heuristikwerte von additiven PDBs summiert werden. Prinzipiell lässt sich aus den Werten zweier beliebiger zulässiger Heuristiken h_1 und h_2 lediglich durch Maximieren $max(h_1(z), h_2(z))$ ein höherer Wert gewinnen, so dass die resultierende Heuristik weiterhin zulässig ist.

Holte et al. (2004) konnten zeigen, dass der Einsatz von n additiven PDBs der Größe m, über deren Heuristikwerte maximiert wird, im Vergleich zur Leistung einer einzelnen additiven PDB der Größe mn zu einer geringeren Knotenexpansion führt. Zur Be-

gründung führten sie an, dass durch das Maximieren der Heuristikwerte die Häufigkeit kleiner Werte reduziert wird. Dies hat zur Folge, dass mehr irrelevante Teilbäume abgeschnitten werden können. Große PDBs beinhalten, aufgrund des größeren Patternraums, zwar größere Heuristikwerte als kleine PDBs. Dieser Vorteil reicht jedoch nicht aus, um die Leistung vieler kleiner zu erreichen. Grund hierfür ist, dass Zustände mit kleinen Heuristikwerten bei der Suche mit IDA* häufiger auftreten als Zustände mit großen Heuristikwerten (vgl. Holte et al., 2004; Korf, 1997). Somit kann durch eine Verringerung der kleinen Heuristikwerte ein größerer Effekt erzielt werden als durch Erhöhen des durchschnittlichen Heuristikwerts.

Das Verwenden von mehreren Heuristiken hat jedoch einen Nachteil: Die Kosten zur Bestimmung des Heuristikwerts steigen mit jeder zusätzlichen Heuristik. Für jede Heuristik muss der Heuristikwert oder – im Falle von PDBs – der Index in das Hauptspeicherarray berechnet werden. Es steigt zwar mit jeder PDB die Wahrscheinlichkeit, irrelevante Teilbäume bei der Suche als solche zu erkennen und abzuschneiden, dieser Effekt sinkt jedoch, je mehr PDBs hinzukommen. Das heißt, es können immer weniger Knoten eingespart werden. Im Extremfall übersteigt die in zusätzliche Heuristikberechnung investierte Zeit die Einsparung: Die Suche dauert länger[4].

2.4.5. Symmetrien im Suchraum

In einigen Problemräumen können Symmetrien im Suchraum ausgenutzt werden, um einen Zustand z in einen anderen Zustand z' umzuwandeln, der die gleiche Distanz zum Ziel hat. Trotzdem kann der Heuristikwert für z' höher sein als für z. Diese Eigenschaft ist deshalb effektiv, weil keine zusätzliche Heuristik für das Nachschlagen von $h(z')$ benötigt wird. Das heißt, mit gleichem Speicheraufwand können zwei Heuristikwerte kombiniert werden. Zwar erhöhen sich damit, wie bei jeder Kombination mehrerer Heuristiken, auch die Kosten pro Knoten. Jedoch lässt sich mit Hilfe von Symmetrien die Zahl der besuchten Knoten stark reduzieren.

Ein Beispiel für die geometrische Symmetrie ist das Spiegeln von Zuständen im Schiebepuzzle. Ein Zustand kann in dieser Domäne unter anderem an der Hauptdiagonalen gespiegelt werden. Neben dieser Spiegelung diskutierten Culberson und Schaeffer

[4]Holte et al. (2004) begegneten diesem Problem, indem sie zum einen, sobald ein für das Abschneiden ausreichend hoher Heuristikwert erreicht wird, aufhören, weitere Heuristiken zu konsultieren. Zum anderen werden durch Umsortierung während des Suchens möglichst leistungsfähige Heuristiken am Anfang befragt. Dennoch müssen im Falle des Nicht-Abschneidens *alle* Heuristiken befragt werden, wodurch die Gesamtgeschwindigkeit der Suche beeinträchtigt wird.

(1998) weitere Möglichkeiten zur Transformation im Schiebepuzzle. Die Spiegelung an der Hauptdiagonalen stellt die am häufigsten verwendete Form dar (vgl. Felner und Adler, 2005; Korf und Felner, 2002; Zahavi et al., 2008). Korf und Felner (2002) konnten dadurch im 15er-Puzzle eine Reduktion der expandierten Knoten um Faktor 3,7 beobachten.

Eine weitere Art von Symmetrien sind duale Zustände (vgl. Zahavi et al., 2008). In der vorliegenden Arbeit wird in der Regel auf das Ausnutzen von Symmetrien verzichtet, im Einzelfall jedoch explizit darauf hingewiesen.

2.5. Zusammenfassung

Die Breitensuche ist besonders für das vollständige Durchsuchen eines Zustandsraums geeignet, hat aber exponentiellen Speicherbedarf. Dagegen benötigt die Tiefensuche lediglich Speicher linear zur Suchtiefe, ist aber weder vollständig noch optimal. Die iterative Tiefensuche kombiniert beide Verfahren, ist vollständig, optimal und benötigt nur linearen Speicher. Frontier Search ist eine Suchtechnik, mit der sich in der Breiten- und Bestensuche Speicher einsparen lässt. Statt einer CLOSED-Liste, wird die Menge der genutzten Operatoren an einem Knoten gespeichert, die im Laufe der Suche zu diesem Knoten geführt haben. Neben der Speicherersparnis erlaubt diese Technik jeden Knoten parallel zu verarbeiten.

Für die zielgerichtete Suche können Heuristiken verwendet werden, um den Suchraum einzuschränken und damit schneller eine Lösung zu finden. PDBs sind sehr leistungsfähige Heuristiken, die in zwei verschiedenen Arten existieren. Vollständige PDBs werden mittels vollständiger Breitensuche des Patternraums einmal berechnet und können dann, bei gleichem Ziel, für alle Probleminstanzen der Domäne genutzt werden. Sie zeichnen sich durch geringe relative Kosten zum Erstellen der Heuristik aus, benötigen jedoch sehr viel Speicher, vor allem wenn sie viele Zustandsvariablen umfassen. IDPDBs werden für jede Probleminstanz neu berechnet. Sie umfassen nur einen kleinen Teil einer vollständigen PDB.

Um die Zahl der besuchten Knoten noch weiter zu verringern, können mehrere Heuristiken verwendet werden. Das Maximum zulässiger Heuristiken bleibt hierbei eine zulässige Heuristik. Im Falle von PDBs können mehrere kleine PDBs zu einem besseren Suchergebnis führen als eine Große.

3. Massiv-parallele Suche mit MapReduce

Bei der zielgerichteten Suche kann eine Heuristik den Problemraum eingrenzen, so dass wesentlich weniger Knoten expandiert werden müssen. PDBs sind sehr leistungsfähige Heuristiken. Dennoch kann die Zahl der besuchten Knoten und mit ihr die benötigte Zeit sehr hoch sein. Zur Lösung schwerer Probleme in angemessener Zeit sind parallele Algorithmen nötig.

Im folgenden Abschnitt wird ein kurzer Überblick über den aktuellen Forschungsstand zur parallelen Suche gegeben. Anschließend wird das MapReduce-Paradigma vorgestellt, welches die Grundlage für die im weiteren Verlauf vorgestellten Algorithmen ist. Diese implementieren mit BFFS eine Kombination der Breitensuche mit Frontier Search, sowie eine Variante zur heuristischen Suche. Mit dem resultierenden Breitensuchalgorithmus MR-BFFS lassen sich erstmals sehr große PDBs erstellen.

3.1. Forschungsstand

Schon früh wurden Ansätze zur parallelen Tiefensuche vorgestellt. Dazu teilten zum Beispiel Kumar und Rao (1990) den Stack der Tiefensuche unter mehreren Prozessoren auf. Hat ein Prozessor den ihm zugewiesenen Teilbaum vollständig durchsucht, versucht er, von einem anderen Prozessor einen weiteren, noch nicht untersuchten Teilbaum zu erhalten. Dieses Schema skaliert in Systemen mit gemeinsamem Speicher beinahe linear. Bei Systemen mit verteiltem Speicher überwiegt jedoch der Overhead für die Kommunikation, so dass in diesen Systemen lediglich ein Speedup[1] von 63 bei 128 Prozessoren erreicht werden kann. Einen anderen Ansatz verfolgten Powley et al. (1990), bei dem auf jedem Prozessor IDA* mit unterschiedlichen Thresholds ausgeführt wird. Um die Optimalität einer gefundenen Lösung zu garantieren, müssen erst alle Prozessoren mit niedrigeren Thresholds ihre Arbeit vollenden. Dieser Ansatz ist aufgrund der geringen Anzahl möglicher Thresholds nur für wenige Prozessoren geeignet. Reinefeld und Schne-

[1] Der Speedup ist eine Kennzahl für die Effizienz eines parallelen Algorithmus. Im Idealfall beträgt er bei N eingesetzten Prozessoren ebenfalls N. In Kapitel 5.2.3 wird der Speedup formal eingeführt.

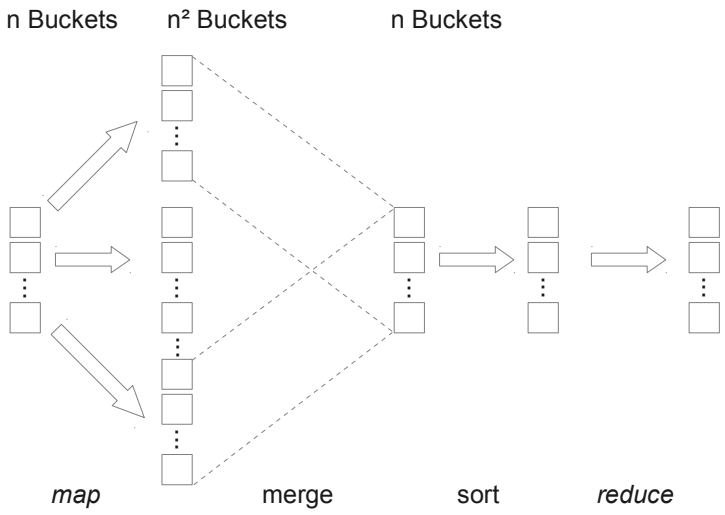


Abbildung 3.1.: Ablauf eines MapReduce Schritts mit n Prozessen

cke (1994) stellten mit AIDA* ein Schema vor, das wie bei Kumar und Rao ebenfalls auf der Aufteilung des Suchraums basiert. Durch lose Synchronisation erreichten sie einen Speedup von 708 bei 1024 Prozessoren.

Neben Ansätzen, die parallele Tiefensuche bzw. iterative Tiefensuche implementieren, existieren von der Breitensuche ebenfalls parallele Varianten. Korf und Schultze (2005) durchsuchten den gesamten Problemraum des 15er-Puzzles mit einer out-of-core BFFS Implementation. Sie nutzten dafür sechs Threads um die Latenz des Fesplattenzugriffs zu verbergen und benötigten insgesamt 680 Stunden. Zhang und Hansen (2006) implementierten Breadth-First Heuristic Search (BFHS) auf einem shared-memory System und nutzten hierfür bis zu 6 Prozessoren.

Nachfolgend werden Algorithmen zur parallelen Breitensuche vorgestellt, die für mehr als 2000 Prozessoren skalieren.

3.2. Das MapReduce-Paradigma

MapReduce ist ein Paradigma zur parallelen Programmierung, das von Dean und Ghemawat (2004) zur massiv-parallelen Datenverarbeitung in großen Clustern eingeführt wurde. Die grundlegende Datenstruktur sind KeyValue-Paare, welche partitioniert in *Buckets* gespeichert werden. Abb. 3.1 verdeutlicht den Ablauf eines MapReduce Schritts. Der Programmierer definiert lediglich die Map- und die Reduce-Funktion.

Zu Beginn werden die n Buckets der Eingabedaten einem oder mehreren Prozessen zugewiesen. Jeder Prozess führt auf allen KeyValue-Paaren eines Buckets die Map-Funktion aus und speichert das Zwischenergebnis in n neue *intermediate*-Buckets. Es hängt sowohl von der Map-Funktion als auch dem einzelnen KeyValue-Paar ab, wie viele neue KeyValue-Paare erzeugt werden. Die Ausgabe der Map-Funktion wird durch eine Hashfunktion $hash$ partitioniert, so dass ein KeyValue-Paar (k, v) genau dann in Bucket i gespeichert wird, wenn $hash(k) \ mod \ n = i$. Im Anschluss führt das MapReduce-Framework die Daten aus allen n^2 Buckets in der *Merge*-Phase wieder so in n Buckets zusammen, dass alle i-ten Ausgabebuckets der Map-Tasks in dem i-ten Bucket der Merge-Phase zusammengefasst werden. Dann werden die Buckets anhand der Schlüsselwerte sortiert. Die darauf folgende Reduce-Funktion fasst alle Paare mit gleichem Schlüssel zusammen und kombiniert die Menge der Values zu einem neuen Value. Das Ergebnis der Kombination ist abhängig von der Aufgabenstellung. Der i-te Reduce-Task verarbeitet von jedem Map-Task nur den i-ten Bucket. Die Ausgabe eines MapReduce-Schritts kann wieder als Eingabe für einen nachfolgenden MapReduce-Schritt dienen.

Anhand des Zählens von Wörtern in einem Text soll dieses Vorgehen exemplarisch dargelegt werden. Hierfür muss der Text zunächst auf die Eingabebuckets aufgeteilt werden. Die Map-Funktion liest dann einen Teil des Textes und emittiert für jedes Wort w das KeyValue-Paar $(w, 1)$. Die Reduce-Funktion erhält, nach der Merge- und Sort-Phase, unter anderem ein oder mehrere KeyValue-Paare mit $(w, 1)$. Die Funktion bildet die Summe über die Werte und erhält so die Häufigkeit des Wortes w im Eingabetext.

Eine Besonderheit ist, dass sowohl in der Map- als auch der Reduce-Phase die Buckets sequenziell gelesen werden (*streaming*). Dies erlaubt den effizienten Einsatz von Festplatten. Alle Map- und alle Reduce-Tasks können parallel ausgeführt werden. Die Reduce-Tasks können jedoch erst starten, wenn alle Map-Tasks beendet sind. Das MapReduce-Framework übernimmt neben dem Zusammenführen und Sortieren Aufgaben wie die Partitionierung der Daten, das Speichermanagement der KeyValue-Paare, die Prozesskommunikation und -synchronisation und implementiert Techniken zur Fehlertoleranz. Fehlgeschlagene Map- oder Reduce-Tasks können zum Beispiel automatisch neu gestartet werden. MapReduce ist sowohl für den Einsatz in großen Computerclustern als auch im Bereich von Multiprozessorsystemen mit gemeinsamem Speicher geeignet (vgl. Dean und Ghemawat, 2004; Ranger et al., 2007).

```
mapper(Position pos, set used_operators) {
  foreach(op ∈ possible_operators(pos))
    if(op ∉ used_operators)
      emit(generate_successor(pos, op), {invert(op)});
}

reducer(Position pos, list<set> operator_list) {
  set used_operators = ∅;
  foreach(o ∈ operator_list)
    used_operators = used_operators ∪ o; //eliminate duplicates
  emit(pos, used_operators);
}

main() {
  front = {(start_pos, ∅)};
  while (front≠∅) {
    intermediate = map(front, mapper());
    front = reduce(intermediate, reducer());
  }
}
```

Abbildung 3.2.: Pseudocode von MR-BFFS.

3.3. Breitensuche und das Erstellen von Pattern Datenbanken

Die Breitensuche eignet sich für das komplette Durchsuchen eines Problemraums. Obwohl dieser möglicherweise Zyklen enthält, wird bei diesem Suchverfahren jeder Knoten genau einmal besucht. Eine Breitensuche in großen Problemräumen erfordert in der Regel den Einsatz von Festplatten, da die Größe der eingesetzten Datenstruktur zum Erkennen von Duplikaten die Kapazität des Hauptspeichers schnell übersteigt.

In diesem Abschnitt wird zunächst ein Algorithmus zur Breitensuche, der auf MapReduce basiert, von Reinefeld und Schütt (2009) vorgestellt. Anschließend wird eine Veränderung diskutiert, die notwendig ist, um in jeder Domäne die Breitensuche durchführen zu können. Diese Verallgemeinerung erlaubt schließlich das Erstellen von PDBs.

3.3.1. Breadth-First Frontier Search mit MapReduce

Dank vollständiger Datenunabhängigkeit aller Knoten innerhalb einer Tiefe ist die Breadth-First Frontier Search (*BFFS*) hervorragend für die parallele Ausführung geeignet. Für die MapReduce-Implementation ist eine Repräsentation der Daten als KeyValue-Paare notwendig: Der Zustand ist der Key und die Menge der bereits genutzten Operationen (vgl. Frontier Search, Kap. 2.3) ist der Value. Die Implementierung von BFFS

mit MapReduce wurde von Reinefeld und Schütt (2009) entwickelt und wird in dieser Arbeit als *MR-BFFS* bezeichnet. Die folgende Aufzählung stellt das Ablaufschema dar:

0. Initialisierung: Füge den Startknoten zur Suchfront hinzu
1. Map: Erzeuge alle Nachfolger, die keine Operationen umkehren und markiere sie mit dem Invertierten der genutzten Operation.
2. Reduce: Fasse alle Duplikate zusammen und markiere sie mit der Vereinigung der Operationen.
3. Fahre bei 1. fort, bis die Suchfront leer ist.

Auf diese Weise kann eine vollständige Breitensuche in einer beliebigen[2] Domäne parallel ausgeführt werden. Einzig der Verzweigungsfaktor des Problems hat eine Auswirkung auf die Parallelität. Der Pseudocode von Mapper, Reducer und des main-Programms umfasst nur wenige Zeilen (siehe Abb. 3.2). Eine Besonderheit des Reducers ist, dass dort eine Technik namens *delayed duplicate detection* (vgl. Korf und Schultze, 2005, S. 1380) implementiert wird. Statt beim Erzeugen der Nachfolger auf Duplikate zu testen, werden diese erst im Reduce-Schritt erkannt und zusammengeführt. Jedes KeyValue-Paar mit gleichen Key – also alle Duplikate – werden durch die Partitionierung einem Bucket zugewiesen. Der Reducer fasst alle Operatoren der Duplikate in eine Liste zusammen und emittiert nur ein neues KeyValue-Paar, das alle Operatoren enthält.

Reinefeld und Schütt (2009) haben MR-BFFS verwendet, um den Suchraum des 15er-Puzzles vollständig zu durchsuchen. Sie nutzten ein distributed-memory System mit 32 Knoten (128 CPUs) und benötigten für die komplette Suche lediglich 66 Stunden.

3.3.2. Erstellen sehr großer PDBs

Der Algorithmus MR-BFFS (Reinefeld und Schütt, 2009) erlaubt eine Breitensuche in Problemräumen mit Zyklen gerader Länge. In Domänen mit ungerader Zyklenlänge können benachbarte Knoten in der selben Tiefe auftreten und werden nicht erkannt. Um dies zu korrigieren, haben wir in Reinefeld et al. (2010) einen Algorithmus vorgestellt, der auch bei ungerader Zyklenlänge korrekt läuft.

Doch zunächst soll das Problem anhand des folgenden Beispiels dargestellt werden: Abbildung 3.3 zeigt einen solchen Graphen mit einem Zyklus der Länge drei. MR-BFFS (Abb. 3.2) auf diesen angewandt, ergäbe folgenden fehlerhaften MapReduce-Ablauf. Die

[2] Dies gilt nur für Suchgraphen mit Zyklen gerader Länge. Für Graphen mit beliebigen Zyklen siehe Abschnitt 3.3.2

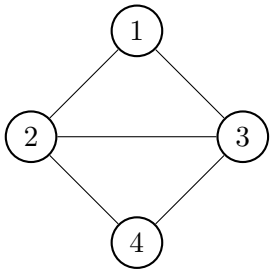


Abbildung 3.3.: Graph mit Zyklus ungerader Länge
von Reinefeld und Schütt (2009)

mittlere Spalte zeigt die Ausgabefront des jeweiligen Schritts. In einem KeyValue-Paar sind der Knoten und die Menge der Vorgänger gespeichert:

Init	$(1, \emptyset)$	
Map	$(2, 1), (3, 1)$	
Reduce	$(2, 1), (3, 1)$	$g = 1$
Map	$(3, 2), (4, 2), (2, 3), (4, 3)$	
Reduce	$(3, 2), (4, \{2, 3\}), (2, 3)$	$g = 2$
Map	$(1, 3), (4, 3), -, (1, 2), (4, 2)$	
Reduce	$(1, \{2, 3\}), (4, \{2, 3\})$	$g = 3$
Map	$\emptyset$	

Das ist kein korrekter Ablauf einer Breitensuche, da die Knoten 2 und 3 nicht als Duplikate erkannt und in Tiefe 2 erneut eingeführt werden. Um dies zu korrigieren, müssen ungerade Zyklen erkannt werden, in denen der Nachfolger eines Knotens zur selben Suchfront gehört. Dazu wird dem Reducer zusätzlich die Ausgabe des vorangegangenen Reduce-Schritts übergeben (dargestellt in eckigen Klammern des nachfolgenden Ablaufs). Der Reducer gibt einen Nachfolger nur dann aus, wenn dieser nicht Teil der letzten Suchfront und somit nicht in der gleichen Tiefe wie der zu expandierende Knoten liegt. Technisch wird das realisiert, indem an jedem Knoten ein zusätzliches Bit gespeichert wird, das anzeigt, ob der Knoten zu einer Suchfront gerader Tiefe gehört oder nicht. Bei der Erzeugung der Nachfolger eines Knotens liest der Mapper das Bit dieses Knotens aus, und speichert in den Nachfolgern das invertierte Bit. Der Reducer stellt einen Zyklus ungerader Länge fest, wenn in den zu reduzierenden Knoten das Bit alterniert. Mit dieser Veränderung produziert MR-BFFS den folgenden korrekten MapReduce-Ablauf:

Init	$(1, \emptyset)$	
Map	$(2,1), (3,1), [(1,-)]$	
Reduce	$(2,1), (3,1)$	$g = 1$
Map	$(3,2), (4,2), (2,3), (4,3), [(2,1), (3,1)]$	
Reduce	$(4, \{2,3\})$	$g = 2$
Map	$\emptyset, [(4, \{2,3\})]$	

Hier erkennt der Reducer in Tiefe 2, dass die Knoten 2 und 3 bereits besucht wurden und gibt diese kein zweites Mal aus. Jeder Knoten wird nur einmal besucht.

Zusätzlich zu der genannten Veränderung in MR-BFFS muss für das Erstellen von additiven PDBs die Kostenfunktion angepasst werden (vgl. Kapitel 2.4.2), denn die Bewegungen von Platzhaltervariablen erhöhen in diesem Fall nicht die Tiefe eines Patterns. Hierzu werden alle benachbarte Knoten, die durch die Veränderung einer Platzhaltervariablen verbunden sind in einem Knoten zusammengefasst.

Mit diesen Mitteln ist es erstmals möglich sehr große PDBs zu erstellen.

3.4. Heuristische Breitensuche mit MapReduce

Die mit MR-BFFS erzeugten PDBs können bei der heuristischen Suche verwendet werden. Im Normalfall wird die PDB zu Beginn des Suchdurchlaufs als Array in den Hauptspeicher geladen, aus dem die Heuristikwerte, während der Suche ausgelesen werden können. Der Algorithmus MR-BFFS wurde allerdings in Hinblick auf sehr große PDBs entwickelt, die größer sein können als der verfügbare Arbeitsspeicher. In diesem Fall wird ein Schema benötigt, das es erlaubt, die PDB bei Bedarf von der Festplatte zu lesen. Im Folgenden wird mit MR-BFIDA* ein Algorithmus vorgestellt, auf dessen Grundlage ein solches Schema möglich wird.

3.4.1. Breadth-First Iterative-Deepening A* mit MapReduce

Breadth-First Iterative-Deepening A* (*BFIDA**) ist eine Variante von IDA* mit breadth-first Expansionsstrategie basierend auf BFFS (vgl. Zhou und Hansen, 2006, S. 395ff). Um aus MR-BFFS eine heuristische Suche machen zu können, sind nur wenige Anpassungen notwendig. Abbildung 3.4 zeigt den für diesen Zweck leicht erweiterten Pseudocode. Der Mapper wird um eine Zeile erweitert, die für das Abschneiden nicht relevanter Teilbäume sorgt. Er gibt einen Nachfolger *succ* nur dann aus, wenn für diesen die von IDA* charakteristische Gleichung $g + h(succ) + 1 <= threshold$ gilt. Der Reducer wird um eine Zeile erweitert, die testet, ob ein Zielknoten gefunden wurde. Im main-Programm wird

```
mapper(Position pos, set used_operators) {
  foreach(op ∈ possible_operators(pos))
    if(op ∉ used_operators) {
      succ = generate_successor(pos, op);
      if(g + 1 + h(succ) <= thresh)
        emit(succ, {invert(op)});
    }
}

reducer(Position pos, list<set> operator_list) {
  set used_operators = ∅;
  foreach(o ∈ operator_list)
    used_operators = used_operators ∪ o; //eliminate duplicates
  solved = (pos == goal_pos);
  emit(pos, used_operators);
}

main() {
  thresh = h(n); solved = false;
  while(!solved) {
    front = {(n, ∅)}; g = 0;
    while (!solved && front ≠ ∅) {
      intermediate = map(front, mapper, thresh, g);
      front = reduce(intermediate, reducer());
      g++;
    }
    thresh++;
  }
}
```

Abbildung 3.4.: Pseudocode von MR-BFIDA*.

eine Schleife eingeführt, die den Threshold inkrementiert. Außerdem muss gestestet werden, ob eine Lösung gefunden wurde (vgl. Reinefeld und Schütt, 2009, S. 332f). Die Implementation von BFIDA* mit MapReduce wird in dieser Arbeit als *MR-BFIDA** bezeichnet.

Wenn der Hauptspeicher ausreicht, um die Heuristik zu laden, kann mit diesem Algorithmus eine parallele heuristische Suche in beliebigen Domänen durchgeführt werden. Andernfalls ist eine Erweiterung des Schemas notwendig, wie sie anschließend vorgestellt wird.

3.4.2. Suche mit großen PDBs

Ist die PDB zu groß für den Hauptspeicher, muss sie bei Bedarf von der Festplatte gelesen werden. Im Algorithmus MR-BFIDA* werden die Heuristikwerte in der Map-Phase

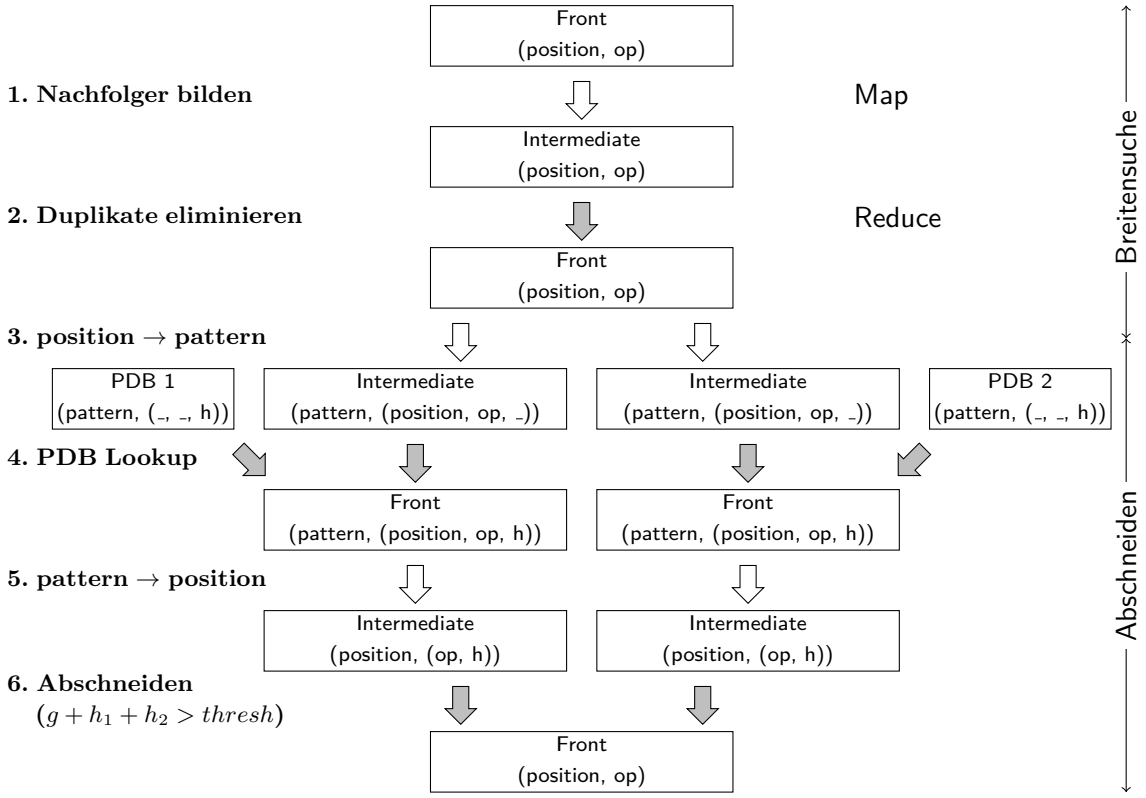


Abbildung 3.5.: Ablaufplan von MR-BFIDA* mit zwei additiven PDBs.

nachgeschlagen. Der Zugriff auf das Array im Hauptspeicher weist dabei eine geringe örtliche Lokalität auf (*random access*). Ein solches Zugriffsmuster ist für Festplatten nicht anwendbar, da es eine ständige Neupositionierung des Lesekopfs erforderlich macht, wobei hohe Latenzzeiten entstehen. Ein wesentlich effizienteres Zugriffsmuster ist das sequenzielle Lesen (*streaming*) einer Datei, wie es in MapReduce zur Anwendung kommt.

Um das effiziente Lesen der PDB von Festplatte zu ermöglichen, sind mehrere MapReduce-Schritte notwendig. Abbildung 3.5 zeigt exemplarisch einen Ablaufplan mit zwei additiven PDBs. Weiße bzw. graue Pfeile stellen Map- bzw. Reduce-Schritte dar. In der ersten Phase wird eine einfache Breitensuche durchgeführt: Die Nachfolger werden erzeugt (1) und anschließend Duplikate eliminiert (2). Der darauf folgende Map-Schritt (3) bildet für jedes KeyValue Paar, also für jedes Paar $(position, op)$, ein Paar der Form $(pattern, (position, op))$ mit dem entsprechenden Pattern für jede PDB. In diesem Schritt wird die Größe der Suchfront verdoppelt, da eine PDB mit zwei Partitionen verwendet

wird. Im Falle von n Partitionen wird die Größe der Suchfront ver-n-facht. Weiterhin werden in diesem Schritt die PDBs der Suchfront hinzugefügt. Im Reduce-Schritt (4) findet das Nachschlagen der Heuristikwerte statt: Nach dem Sortieren der Front ist der Reducer in der Lage, dem Pattern einen Heuristikwert zuzuweisen. Überflüssige Heuristikwerte, d.h. solche, denen keine Nachfolger zugeordnet werden können, werden an dieser Stelle verworfen. Der anschließende Map-Schritt (5) entfernt die Pattern aus den KeyValue Paaren, behält aber die gewonnenen h Werte aus der Heuristik. Der letzte Reduce-Schritt (6) schneidet alle Knoten ab, deren geschätzte Kosten zum Zielknoten größer sind als der aktuelle Grenzwert: $g + h_1 + h_2 > thresh$. Diese Schritte werden wiederholt, bis ein Zielknoten gefunden wurde. Durch dieses Schema können PDBs für die heuristische Suche genutzt werden, die mit ihrer Gesamtgröße nicht in den Hauptspeicher passen.

3.5. Das MR-Search Framework

Im Rahmen dieser Arbeit entstand das MapReduce Framework *MR-Search*. Es ist für Breitensuche und heuristische Suche in großen Graphen ausgelegt. MR-Search ist eine in C++ geschriebene Klassenbibliothek, die aus mehreren Schichten besteht, dargestellt in Abbildung 3.6.

Kernelement von MR-Search ist ein generisches MapReduce-Framework, das verschiedene Strategien zur Parallelisierung und zur Datenhaltung unterstützt. Neben Berechnungen im Hauptspeicher (*in-memory*) sind ebenfalls out-of-core Berechnungen möglich, bei denen (Zwischen-)Ergebnisse auf dem Sekundärspeicher abgelegt werden. Letzteres wird vor allem für das Erstellen von PDBs verwendet, während andere Aufgaben, wie die heuristische Suche, typischerweise im Hauptspeicher stattfinden. Out-of-core Berechnungen können auf parallelen Dateisystemen wie Lustre (vgl. Schwan, 2003) oder XtreemFS (vgl. Hupfeld et al., 2008) sowie auf lokalen Festplatten ausgeführt werden.

Zur Parallelisierung werden Systeme mit gemeinsamem Speicher (*shared-memory*) mittels OpenMP und Systeme mit verteiltem Speicher (*distributed shared-memory*) durch MPI unterstützt. Die Parallelisierung mit OpenMP ist einfach. Sowohl in der Map- als auch in der Reduce-Phase wird die Map- bzw. Reduce-Funktion durch Schleifen über alle Partitionen ausgeführt. Diese Schleifenläufe sind mit der `parallel for` Direktive von OpenMP parallelisiert. Die Merge-Phase entfällt für diesem Fall, da alle Threads auf den selben Speicher zugreifen.

Der Austausch der Daten nach der Map-Phase ist jedoch notwendig, wenn MapRedu-

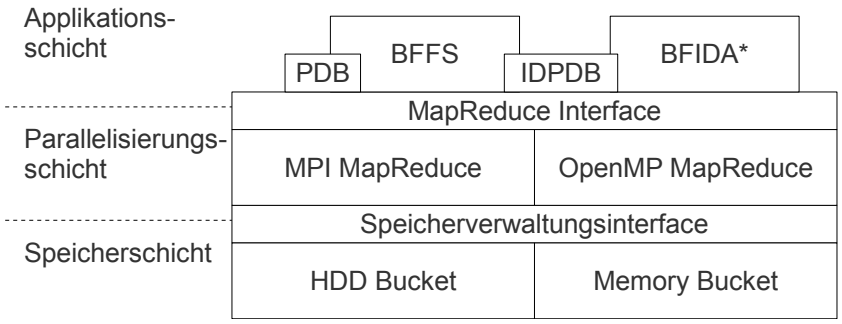


Abbildung 3.6.: Die Schichtarchitektur von MR-Search

ce mit MPI ausgeführt wird. Für diese Aufgabe enthält jeder Bucket eine Routine, die von der MPI MapReduce Komponente gestartet wird. An dieser Stelle muss zwischen out-of-core und in-memory Berechnungen unterschieden werden. Im Falle der out-of-core Berechnung werden die Daten mittels HTTP zwischen den Rechnerknoten ausgetauscht, sofern kein paralleles Dateisystem vorliegt. Jeder Bucket ist dabei in einer Datei gespeichert. Die in-memory Variante ist deutlich komplexer. Hierbei werden die Daten mit dem MPI-Befehl `MPI_Alltoallv` übertragen, der das Austauschen ungleich großer Datenmengen zwischen den Prozessen erlaubt. Dennoch müssen, im Gegensatz zur HTTP-Variante, die exakten Größen der Partitionen bestimmt werden, bevor die eigentlichen Daten übertragen werden können. Das führt zu einem zweiten Aufruf des besagten Befehls.

Der Zugriff auf das MapReduce-Framework ist durch ein Interface standardisiert. Dieses kann von beliebigen, auf MapReduce basierenden Algorithmen genutzt werden. Der Fokus dieser Arbeit und der von MR-Search liegt auf dem Schiebepuzzel. Das Framework unterstützt Zustandsrepräsentationen und -operationen für das 8er-, 15er-, 24er-, und 35er-Puzzle. Weitere Domänen für zukünftige Forschungsarbeit können hinzugefügt werden, ohne Anpassungen an den Graphenalgorithmen oder dem MapReduce-Framework vorzunehmen. In MR-Search sind die vorgestellten Graphenalgorithmen BFIDA*, BFFS sowie Varianten davon zur Erstellung von PDBs und IDPDBs implementiert. Diese sind mit allen beschriebenen MapReduce Varianten kompatibel. Zur Suche können sämtliche in Abschnitt 2.4 vorgestellten Heuristiken verwendet werden. Es sind sowohl Implementationen für einfache Heuristiken wie Manhattan Distanz und Linear Conflicts als auch für die Nutzung von PDBs und IDPDBs vorhanden.

3.6. Zusammenfassung

Die Besonderheit des MapReduce-Paradigmas ist dessen Generalität. Der Programmierer muss sich lediglich um den domänenspezifischen Code wie, im Falle der Graphensuche, Zustandsrepräsentation und Nachfolgergenerierung kümmern. Sämtliche Aufgaben zur parallelen Verarbeitung übernimmt das MapReduce-Framework.

Mit MR-BFFS wurde ein von Reinefeld und Schütt (2009) auf MapReduce angewandter Algorithmus zur Breitensuche eingeführt. Der Algorithmus erlaubt eine Breitensuche in Domänen mit Zyklen gerader Länge. Um eine Breitensuche auch in Domänen mit Zyklen ungerader Länge durchführen zu können, wurde darauf folgend eine Anpassung des Algorithmus vorgenommen, so dass sehr große PDBs erstellt werden können. Die Ergebnisse sind in Reinefeld et al. (2010) veröffentlicht worden.

Diese Implementation von BFIDA* auf MapReduce geht auf Reinefeld und Schütt (2009) zurück. Darauf basierend wurde ein Ablaufplan entwickelt, mit dem sehr große PDBs verwendet werden können, bei denen das Laden in den Hauptspeicher nicht möglich ist.

Zuletzt wurde das im Rahmen dieser Arbeit entwickelte MapReduce-Framework MR-Search vorgestellt. Es erlaubt verschiedene Arten der Parallelisierung und der Datenhaltung. Obwohl es zur Zeit für Schiebepuzzle ausgelegt ist, kann es für neue Forschungszwecke erweitert werden.

4. Effiziente Heuristiken – Eine empirische Analyse

Nachdem im vorangegangenen Kapitel ein Algorithmus vorgestellt wurde, der es erstmals erlaubt, sehr große PDBs zu erstellen, wird nun unter anderem die Effizienz solcher PDBs untersucht. Dieses Kapitel geht der Frage nach, wie der für eine Heuristik zur Verfügung stehende Hauptspeicher möglichst effizient genutzt werden kann. Im Allgemeinen ist die Leistung einer PDB umso höher, je mehr Elemente der Permutation sie umfasst, wobei gleichzeitig der durch die PDB belegte Hauptspeicher steigt (vgl. Holte und Hernádvölgyi, 1999; Korf, 1997). In Abbildung 4.1 sind die Leistungen aller aus zwei Partitionen bestehenden additiven PDBs (*dual-additive* PDBs) abgebildet. Daraus ist ersichtlich, dass eine größere PDB nicht zwangsläufig besser ist. So gibt es beispielsweise nur drei 5+3 PDBs, die besser sind als die beste 4+4 PDB. Diese Überlappung besteht sowohl bei additiven PDBs (vgl. Reinefeld et al., 2010) als auch bei nicht-additiven PDBs (vgl. Hernádvölgyi und Holte, 2004).

Die bisherige Forschung konzentrierte sich auf die Analyse der Leistung einzelner PDBs bzw. additiver PDBs. Der Beitrag von Holte et al. (2004) hat gezeigt, dass durch Maximieren der Heuristikwerte n additive PDBs der Größe m eine bessere Suchleistung aufweisen können als eine additive PDB der Größe nm. Im Anschluss wird analysiert, wie sich mehrere PDBs mit unterschiedlicher Größe verhalten und welche Faktoren bei solchen Konstellationen zum Erfolg der Gruppe beitragen. Dazu werden in einer Untersuchung größenheterogene Gruppen für das 8er-Puzzle gebildet und analysiert. Das 8er-Puzzle wurde gewählt, weil diese Domäne klein genug ist, um eine erschöpfende Untersuchung aller dual-additiver PDBs zu ermöglichen.

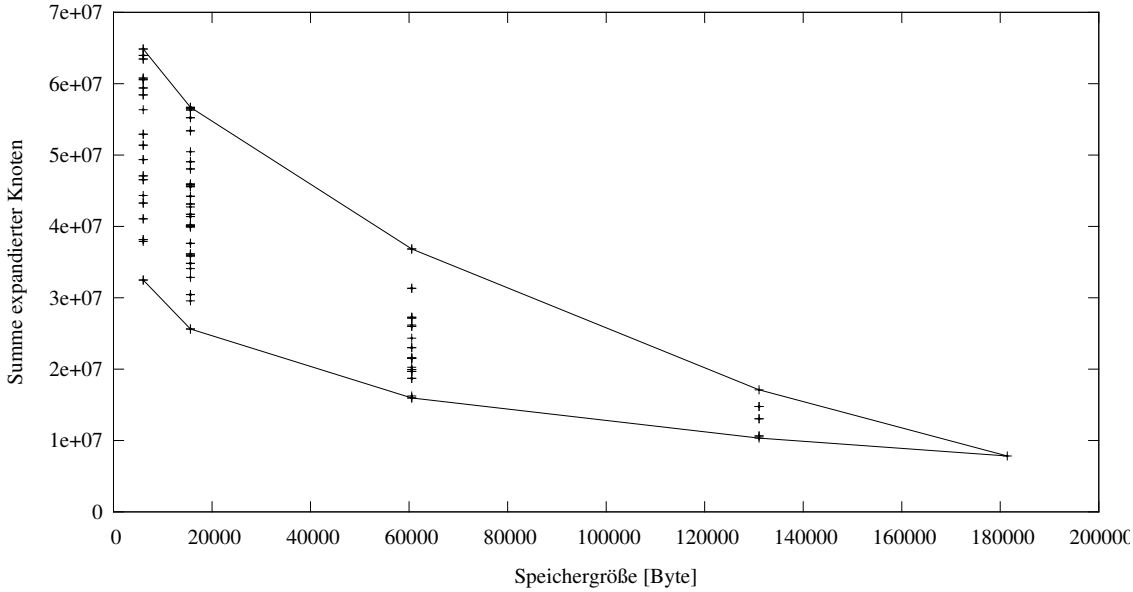


Abbildung 4.1.: Leistung aller additiver PDBs im 8er Puzzle mit zwei Partitionen gegenüber dem Speicherbelegung. PDBs von links nach rechts: 4+4, 5+3, 6+2, 7+1, 8+0.

4.1. Begrifflichkeiten und analytische Vorannahmen

Jede PDB h hat eine spezifische Größe $S(h)$, die sie im Hauptspeicher belegt. Für eine PDB eines 4-Patterns im 8er-Puzzle sind beispielsweise 3024 Bytes nötig[1]. Eine additive 5+3 PDB belegt $15120 + 504 = 15624$ Bytes. Die Zahl der möglichen dual-additiven PDBs ist in der ersten Zeile von Tabelle 4.1 dargestellt[2]. Weiterhin gibt die Tabelle die Größen absolut und relativ zur Größe des Problemraums an.

Ein zentraler Begriff der Untersuchung ist die Effizienz einer Heuristik im Hinblick auf den benötigten Hauptspeicher. Dazu ist die Definition grundlegender Begriffe notwendig:

- Eine *Gruppe* ist eine Menge P von PDBs, deren Heuristikwerte bei der Suche maximiert werden. Die *Gruppengröße* gibt die Zahl der PDBs $|P|$ in der Gruppe an.

[1] Diese und alle folgenden Größen beziehen sich auf PDBs, die mittels compact-Mapping gespeichert und nach der Position des Blanks komprimiert sind. Vgl. hierzu Kapitel 2.4.2

[2] Die Zahl der möglichen n+m PDBs entspricht $\binom{8}{n}$. Bei $n = 4$ jedoch nur die Hälfte. Vgl. die neunte Zeile des Pascalschen Dreiecks.

PDB	4+4	5+3	6+2	7+1	8+0
Anzahl	35	56	28	8	1
Größe	6048	15624	60552	131049	181440
rel. Größe	3%	9%	33%	72%	100%

Tabelle 4.1.: Größen und Anzahl aller dual-additiver PDBs des 8er-Puzzles.

- Die *Speicherbelegung* $S(G)$ einer Gruppe G ist die Summe der Größen der einzelnen PDBs. $S(G) = \sum_{h \in G} S(h)$.
- Neben der Leistung wird im Folgenden die Zuverlässigkeit einer Heuristik betrachtet. Eine Heuristik ist *zuverlässig*, wenn die Schwankung ihrer Leistung gering ist[3]. Um die Leistung über mehrere Probleme vergleichbar machen zu können, wird ein Referenzmaß benötigt. Für jedes gelöste Problem p wird der Leistungsverlust $L = \frac{K_h(p)}{K_{h^*}(p)}$ relativ zur perfekten Heuristik berechnet, der angibt, wie viel Prozent Knoten durch h zusätzlich expandiert werden mussten. Der Variationskoeffizient[4] des Leistungsverlusts dient als Maß für die Schwankung der Leistung.
- Sei H eine Menge von Heuristiken, P eine Menge von Probleminstanzen, A ein Suchalgorithmus und $K_h(P),\ h \in H$ die Summe der expandierten Knoten, die A unter Verwendung von h für das Lösen aller Probleminstanzen benötigt. Die Heuristik h_1 ist genau dann *effizienter als* h_2, wenn $S(h_1) \leq S(h_2)$ und $K_{h_1}(P) < K_{h_2}(P)$.
- Wenn keine Heuristik $h \in H$ existiert, so dass h effizienter ist als h_1, dann ist h_1 *effizient bzgl. H.*

Es folgen zuerst drei Hypothesen zum Verhalten der Gruppen, bevor die Untersuchung vorgestellt wird.

a. Die erste Annahme ist, dass sich die negative Korrelation zwischen der Größe einer PDB und der Zahl der expandierten Knoten auf eine Gruppe in Bezug auf die Gruppengröße übertragen lässt. Je mehr PDBs somit in einer Gruppe zusammengefasst sind, desto geringer ist die Zahl der expandierten Knoten.

[3] Die Zuverlässigkeit einer Heuristik ist besonders dann interessant, wenn nur wenige Probleme gelöst werden sollen. Andernfalls interessiert nur die durchschnittliche Leistung.

[4] Der Variationskoeffizient ist definiert als $V = \frac{\sigma}{\bar{X}}$, also der Standardabweichung relativ zum arithmetischen Mittelwert.

Als Begründung dient die Bedingung $g + max(h_1, ..., h_n) > threshold$, mit der bei der Suche Teilbäume abgeschnitten werden. Jeder Heuristikwert h_i kann als Zufallsvariable aufgefasst werden. Demzufolge steigt mit jeder weiteren PDB die Chance, dass der Heuristikwert weiter erhöht wird und somit die Chance, dass die Gesamtkosten den Threshold übersteigen. Die Untersuchung von Holte et al. (2004) bestätigt diesen Effekt.

b. Da größere PDBs im Allgemeinen besser sind, wird erwartet, dass Gruppen mit großen PDBs effizienter sind.

   Zur Erklärung wird das Spektrum der Heuristikwerte in PDBs betrachtet. Durch eine Gruppe kleiner PDBs wird die Wahrscheinlichkeit reduziert, einen kleinen Heuristikwert zu erhalten. Die Heuristikwerte dieser Gruppe sind jedoch nach oben hin begrenzt. In großen PDBs sind größere Heuristikwerte gespeichert, so dass in der Gruppe sowohl das Verringern der kleinen Heuristikwerte als auch das Erhöhen des durchschnittlichen Heuristikwerts zu einer verringerten Knotenexpansion führt.

c. Der letzte Punkt betrifft die Zuverlässigkeit der Heuristik. Es wird angenommen, dass die Leistungsschwankung für größere Gruppen abnimmt.

   Einzelne PDBs weisen eine große Schwankung auf, da eine PDB nicht für jeden Teil des Suchraums gleich gut ist. In einer Gruppe kann die lokale Schwäche der einen PDB durch die Stärke einer anderen ausgeglichen werden.

4.2. Zentrale Ergebnisse

In Reinefeld et al. (2010) haben wir die Leistung aller 127 möglichen dual-additiven PDBs des 8er-Puzzles analysiert. Für die Untersuchung wurden Teilmengen der 127 dual-additiven PDBs mit 1 bis 10 Elementen gebildet. Gruppen, deren Speicherbelegung den der perfekten Heuristik (181440 Bytes) überschritten, wurden von der Analyse ausgeschlossen. Für die Untersuchung wurden mit jeder Gruppe alle Positionen des 8er-Puzzles mit BFIDA* gelöst. Die im Schiebepuzzle mögliche Spiegelung um die Hauptdiagonale wurde dabei nicht berücksichtigt, da sonst jede PDB bereits eine implizite Gruppe von zwei PDBs darstellen würde[5].

[5] Details zu Aufbau und der Durchführung des Versuchs sind dem methodischen Anhang A.1 zu entnehmen.

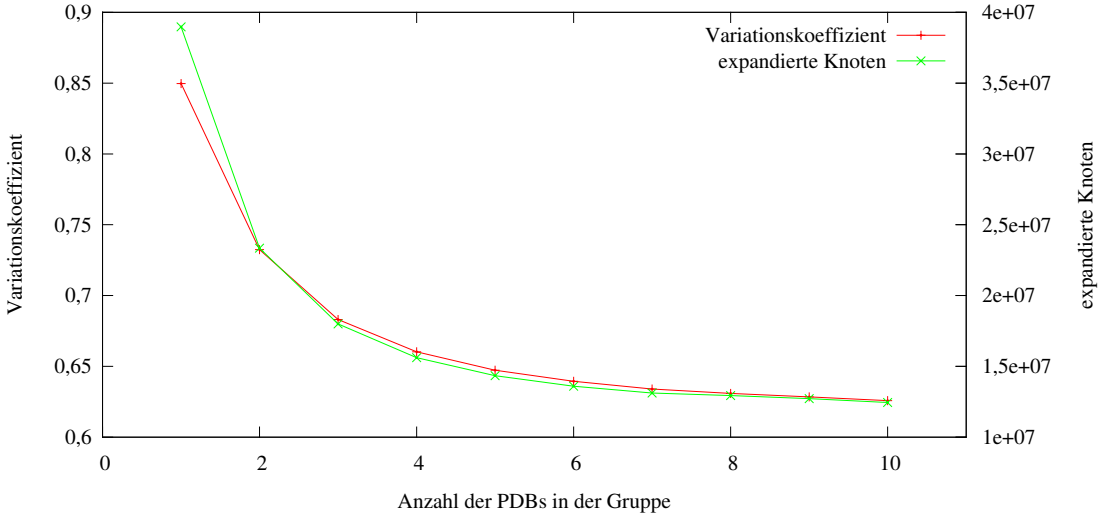


Abbildung 4.2.: Variationskoeffizient des Verlusts gegenüber der perfekten Heuristik und durchschnittliche Summe expandierter Knoten nach Gruppengröße.

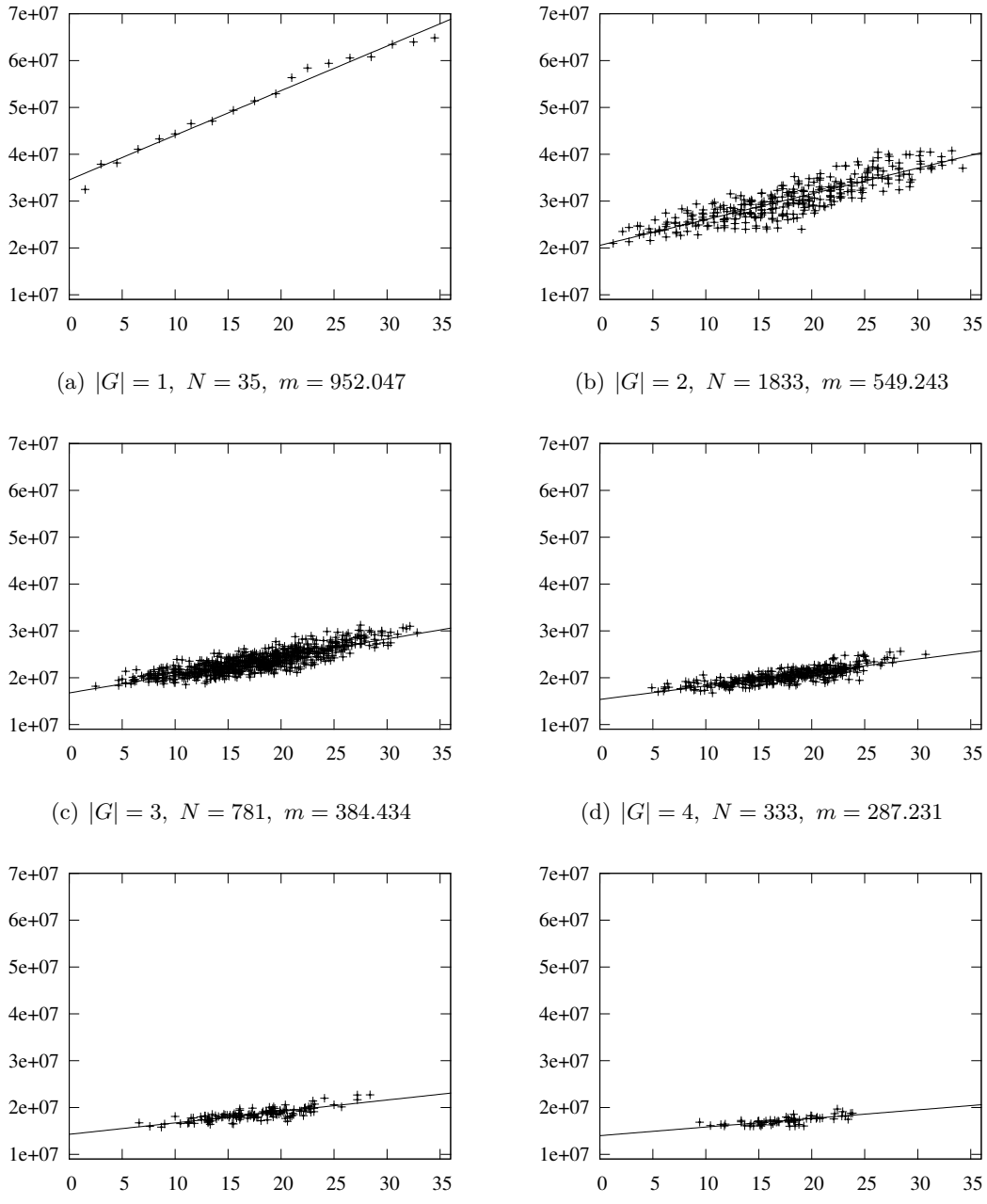
4.2.1. Einfluss der Gruppengröße

Zunächst wird der Zusammenhang zwischen der Gruppengröße und der Zuverlässigkeit der Leistung untersucht. Abbildung 4.2 zeigt den durchschnittlichen Variationskoeffizienten des Verlusts gegenüber der perfekten Heuristik zur Gruppengröße. Der Variationskoeffizient der Leistung nimmt mit Hinzunehmen weiterer PDBs ab und die Zuverlässigkeit der Leistung wird erhöht. Dies bestätigt Annahme c. Während mit einer PDB die Leistung durchschnittlich um 74,9% des Mittelwerts schwankt, kann die durchschnittliche Schwankung mit acht oder mehr PDBs um 13% gesenkt werden.

Ferner wird in Abbildung 4.2 die durchschnittliche Anzahl der expandierten Knoten zur Gruppengröße dargestellt. Der Verlauf der Knotenzahl gleicht stark dem des Variationskoeffizienten. Mit steigender Gruppengröße nimmt die Zahl der expandierten Knoten ab, wobei dieser Effekt zunehmend schwächer wird. Dies verifiziert Hypothese a.

Nun wird der Einfluss der individuellen Leistung einer PDB auf die Leistung der Gruppe untersucht. Dazu wurden alle 35 4+4 PDBs in eine Rangfolge gemäß Knotenexpansion gebracht[6]. Abbildung 4.3 zeigt für die Gruppengrößen $|G|$ von 1 bis 6 die Summe der expandierten Knoten zum durchschnittlichen Rang der Gruppe, sowie die jeweilige Regressionsgerade. An den Teilbildern sind die Gruppengröße, die Zahl der Gruppen die-

[6] Erläuterungen zur Rangbildung im methodischen Anhang A.1.



(a) $|G| = 1,\ N = 35,\ m = 952.047$

(b) $|G| = 2,\ N = 1833,\ m = 549.243$

(c) $|G| = 3,\ N = 781,\ m = 384.434$

(d) $|G| = 4,\ N = 333,\ m = 287.231$

(e) $|G| = 5,\ N = 126,\ m = 244.333$

(f) $|G| = 6,\ N = 62,\ m = 183.614$

Abbildung 4.3.: Summe der expandierten Knoten (Y-Achse) zu durchschnittlichem Rang der Gruppe (X-Achse) für Gruppen von 4+4 PDBs sowie Regressionsgerade. Gruppengröße $|G|$, Fälle N, Steigung der Regressionsgeraden m. Der Einfluss des individuellen Rangs ist mit steigender Gruppengröße zunehmend unbedeutend für die Leistung der Gruppe.

ser Größe sowie die Steigung der Regressionsgeraden notiert. Abbildung 4.3(a) zeigt die individuelle Leistung der 4+4 PDBs in Verbindung mit ihrem Rang. In den Abbildungen (b) bis (f) ist zu erkennen, dass der durchschnittliche Rang und damit die individuelle Leistung der PDBs in der Gruppe zunehmend weniger Einfluss auf die Leistung der Gruppe hat. Der Einfluss des durchschnittlichen Rangs wird beim Übergang von einer zu zwei PDBs um 42% reduziert und sinkt für Gruppen mit sechs PDBs um weitere 38%. Für Gruppen mit vielen PDBs spielt der individuelle Rang kaum eine Rolle.

4.2.2. Einfluss der Gruppenzusammensetzung

Es werden die Einflussfaktoren Speicherbelegung S und Zusammensetzung der Gruppen auf die Summe der expandierten Knoten K untersucht. Für jede Speicherbelegung wurde der Durchschnitt $\bar{K}(S)$ der Summen expandierter Knoten gebildet. Aussagen über die Effizienz beziehen sich auf die Menge der durchschnittlichen Gruppen[7]. Der Zusammenhang von Speicherbelegung S und der durchschnittlichen Leistung $\bar{K}(S)$ ist in Abbildung 4.4 dargestellt. Dabei wird eine negative Korrelation zwischen S und $\bar{K}(S)$ deutlich: Je größer eine Gruppe ist, desto weniger Knoten werden expandiert. Der Effekt nimmt mit zunehmender Speicherbelegung ab. Unter Berücksichtigung der Gruppenzusammensetzung lassen sich weitere Aussagen treffen. In der Abbildung werden die Datenpunkte nach den jeweils größten PDBs einer Gruppe differenziert. Die Legende benennt jede Gruppe entsprechend der Anzahl und des Typs der größten PDBs einer Datenreihe, so dass beispielsweise Gruppen der Datenreihe „1x 6+2, 1x 5+3“ als größte PDBs eine 6+2 und eine 5+3 PDB enthalten. Restklassen wie „>= 4x 5+3“ enthalten vier oder mehr 5+3 PDBs. Die verbleibende Unbekannte der Gruppenzusammensetzung ist die Anzahl der 4+4 PDBs. Außerdem sei auf den einzelnen Datenpunkt bei 181440 Bytes hingewiesen, der die Leistung K_{h^*} der perfekten Heuristik repräsentiert. Die Abbildung ist weiterhin bei 6048, 15624, 60552 und 131049 Bytes durch vertikale Linien unterteilt, welche die Verfügbarkeit der nächst größeren PDBs markieren (vgl. Tab. 4.1).

Bei Betrachtung der Gruppenzusammensetzung kann unter anderem das Ergebnis von Holte et al. (2004) verifiziert werden, dass eine Gruppe mehrerer kleiner PDBs weniger Knoten expandieren als eine einzelne große PDB. Die durchschnittliche Leistung von Gruppen mit 4+4 PDBs ist, bei vergleichbarer Speicherbelegung, deutlich besser als die mittlere Leistung einer einzelnen 5+3 PDB. Ähnlich verhält es sich mit der durch-

[7] Hier wird die durchschnittliche Leistung der Gruppen mit der Leistung einer durchschnittlichen Gruppe synonym verwendet.

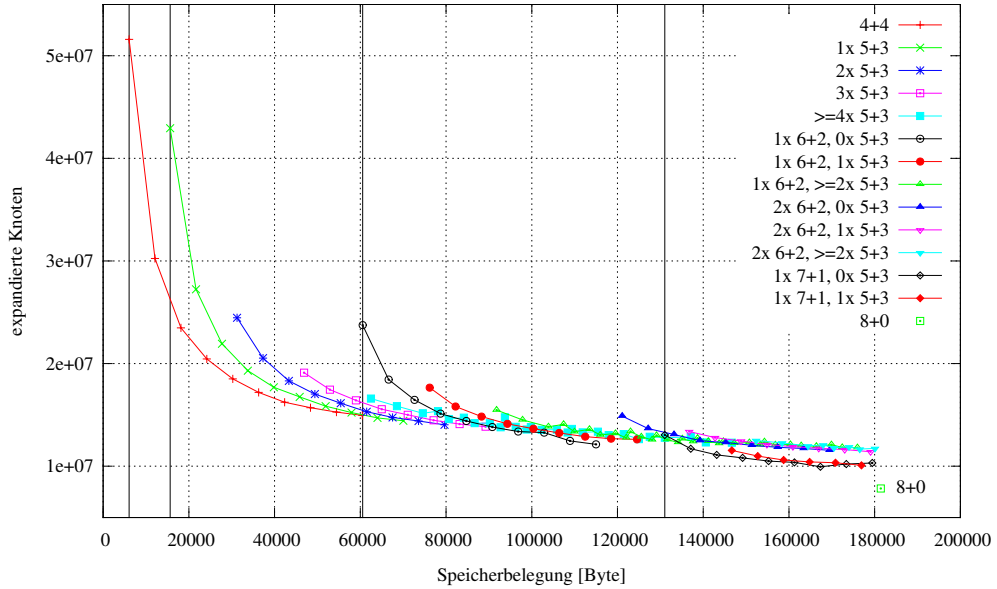


Abbildung 4.4.: durchschnittliche Summe expandierter Knoten nach Speicherbelegung. Keine Ausnutzung von Symmetrien.

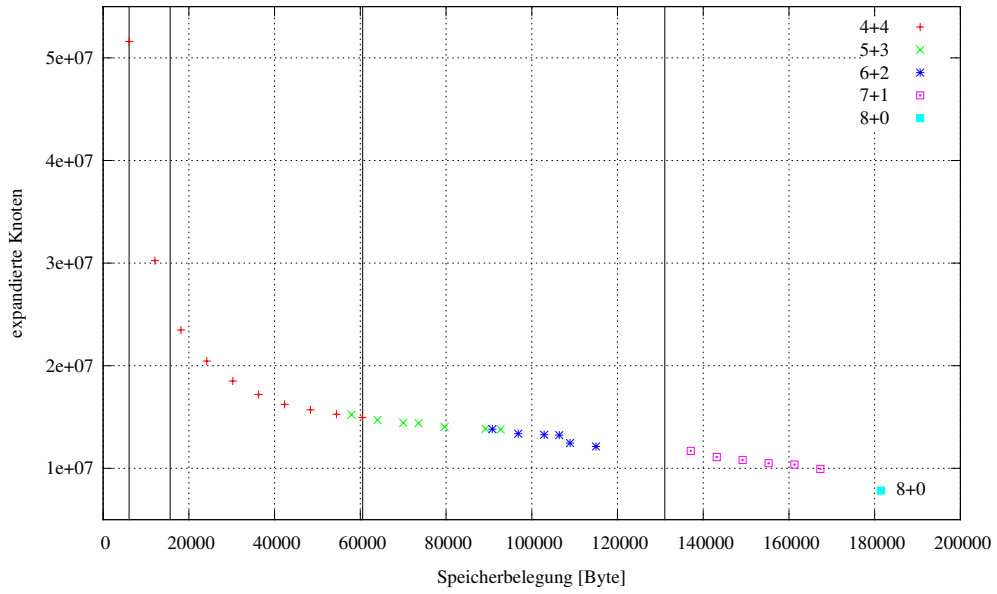


Abbildung 4.5.: durchschnittliche Summe expandierter Knoten nach Speicherbelegung. Nur der effiziente Rand ist dargestellt. Keine Ausnutzung von Symmetrien

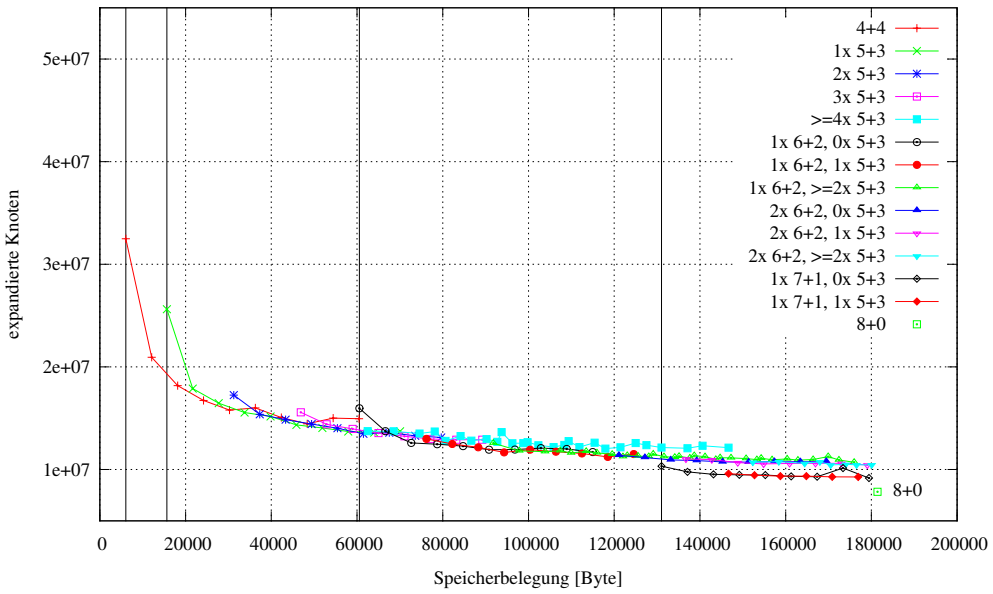


Abbildung 4.6.: kleinste Summe expandierter Knoten nach Speicherbelegung. Keine Ausnutzung von Symmetrien.

schnittlichen Leistung größerer PDBs. Die Differenz zu den effizienten Gruppen nimmt allerdings mit zunehmender Speicherbelegung ab. Einzelne große PDBs sind demnach nicht effizient, aber durch Hinzunehmen weiterer kleiner PDBs steigert sich die Leistung. Beispielsweise ist eine Gruppe bestehend aus einer 6+2 PDB und fünf 4+4 PDBs effizient. In Abbildung 4.5 sind ausschließlich die effizienten Gruppen in Bezug auf die durchschnittliche Leistung dargestellt. Diese bilden den *effizienten Rand*. In der Abbildung wird nur noch nach dem Typ der größten PDB unterschieden. Aus der Darstellung geht erstens hervor, dass Gruppen mit 4+4 PDBs sehr effizient sind. Erst bei knapp 58000 Bytes erscheinen Gruppen mit 5+3 PDBs auf dem effizienten Rand. Letztendlich werden für effiziente Gruppen mit großer Speicherbelegung große PDBs benötigt.

Analog zu Abbildung 4.4 zeigt Abbildung 4.6 nun die Leistung der besten Gruppen je Speicherbelegung S. Dazu wurde das Minimum $min(K(S))$ der Summen expandierter Knoten bestimmt. Die Kernaussagen für diese Betrachtung stimmen im Wesentlichen mit denen der mittleren Leistung überein: Erstens expandieren Gruppen mit größerer Speicherbelegung weniger Knoten. Zweitens sind einzelne PDBs ineffizient und drittens expandieren Gruppen mit großen PDBs, bei gleicher Speicherbelegung, weniger Knoten. Für die besten Gruppen gibt es jedoch zwei Unterschiede. Zum einen expandiert die

beste 7+1 PDB weniger Knoten als vergleichbare Gruppen mit kleineren PDBs. Zum anderen wurde für die durchschnittliche Leistung festgestellt, dass eine bestimmte Gruppengröße notwendig ist, damit eine Gruppe effizient ist. So sind für die durchschnittliche Leistung einer Gruppe mit einer 6+2 PDB fünf zusätzliche 4+4 PDBs nötig, damit die Gruppe effizient ist. Für die beste Leistung werden dagegen nur zwei zusätzliche 4+4 PDBs benötigt, so dass die benötigte Gruppengröße bei Betrachtung der besten Leistung deutlich geringer ist.

4.3. Zusammenfassung

Die vorgestellte Analyse zeigt das Verhalten von Gruppen mit unterschiedlich großen PDBs in der Domäne des 8er Puzzles.

Es konnte gezeigt werden, dass mit steigender Gruppengröße zunehmend weniger Knoten expandiert werden, die Leistung zuverlässiger wird und zudem der Einfluss der individuellen Leistung auf die Leistung der Gruppe reduziert wird. Außerdem wurde der Einfluss der Speicherbelegung der Gruppe und die Gruppenzusammensetzung auf die Zahl er expandierten Knoten analysiert. Einzelne PDBs sind ineffizient, doch durch Hinzunahme weiterer PDBs werden diese letztendlich effizient. Bei den besten Gruppen ist dafür eine geringere Gruppengröße notwendig als im Durchschnitt. Die Untersuchung stellte heraus, dass der Einsatz von großen PDBs sinnvoll ist, wenn der zur Verfügung stehende Hauptspeicher ausreicht, eine Gruppe zusammen mit einigen kleineren PDBs zu bilden.

Eine Verifikation der Ergebnisse für andere Domänen bleibt an dieser Stelle offen. Zukünftige Untersuchungen könnten sich beispielsweise den Domänen Rubik's Cube, Türme von Hanoi oder dem Pancake Puzzle widmen.

5. Anwendung der vorgestellten Algorithmen auf große Probleme

In diesem Kapitel werden die Algorithmen MR-BFFS und MR-BFIDA* auf große Probleme angewandt. Hierzu wird das 24er-Puzzle als Domäne gewählt.

Der erste Abschnitt widmet sich dem Erstellen von PDBs. Dazu werden zehn 6+6+6+6 PDBs vorgestellt, die für die Suche eingesetzt werden. Außerdem wird die Erzeugung einer vollständigen 8+8+8 PDB dokumentiert. Im zweiten Teil werden die heuristische Suche mit BFIDA* beziehungsweise Eigenschaften der MapReduce-Implementation MR-BFIDA* untersucht und die Suchverfahren BFIDA* und IDA* hinsichtlich der besuchten Knoten verglichen. Anschließend wird die Leistung der erstellten PDBs betrachtet. Da das vierte Kapitel dieser Arbeit zeigt, dass Gruppen mit wenigen großen und mehreren kleinen PDBs effizient sind, werden an dieser Stelle ebenfalls Gruppen von Heuristiken gebildet. Danach wird die Skalierbarkeit von MR-BFIDA* und die Anwendbarkeit des im dritten Kapitel vorgestellten Schema zur Verwendung von MR-BFIDA* mit PDBs von Festplatte untersucht.

5.1. PDBs erstellen mit MR-BFFS

Mit dem in Kapitel 3.3 vorgestellten Algorithmus MR-BFFS ist es möglich sehr große PDBs zu erstellen. Es werden PDBs für das 24er-Puzzle erzeugt, die anschließend für die heuristische Suche verwendet werden.

5.1.1. Eine vollständige 8+8+8 PDB

Um die Leistungsfähigkeit von MR-BFFS zu demonstrieren, wird eine vollständige 8+8+8 PDB erzeugt. Dafür wurde aus den in Felner und Adler (2005) angegebenen PDBs die in Abbildung 5.1 dargestellte Partitionierung gewählt.

Jede Partition der 8+8+8 PDB belegt 40 GByte Speicher, so dass die gesamte Heuristik 120 GByte benötigt. Die Suchfronten für das Erstellen der Datenbank sind jedoch

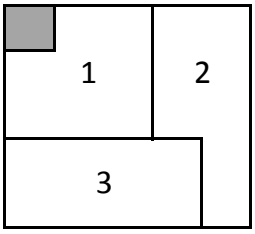


Abbildung 5.1.: 8+8+8 Partitionierung des 24er-Puzzles aus Felner und Adler (2005).

Tiefe	Zustände	Tiefe	Zustände
0	1	24	6.417.787.592
1	2	25	7.927.746.840
2	4	26	9.070.115.230
3	18	27	9.617.642.082
4	104	28	9.434.335.505
5	490	29	8.576.519.764
6	2.086	30	7.214.079.806
7	8.186	31	5.625.300.258
8	29.463	32	4.054.430.789
9	100.060	33	2.704.618.046
10	315.889	34	1.661.055.971
11	944.422	35	939.594.234
12	2.636.994	36	484.977.426
13	6.994.492	37	228.583.360
14	17.505.912	38	97.316.859
15	41.751.488	39	37.499.774
16	94.017.603	40	12.898.174
17	200.556.334	41	4.027.624
18	401.724.084	42	1.152.493
19	756.713.370	43	299.942
20	1.332.374.499	44	71.172
21	2.195.428.254	45	14.928
22	3.371.141.432	46	2.112
23	4.826.085.376	47	96
		Σ	87.358.400.640

Tabelle 5.1.: Zustände als Funktion der Tiefe im Suchgraph des 8-Pattern (1) aus Abb. 5.1

wesentlich größer als die resultierende PDB, weshalb die Berechnung out-of-core durchgeführt werden muss.

Die PDB wurde auf 31 Rechnenknoten eines Clusters erstellt, die je mit zwei AMD Quad-Core (2,3 GHz), 8 GByte RAM und einer Festplatte mit 1 TByte ausgestattet sind. Die Knoten sind mit einem Gigabit Ethernet Netzwerk verbunden. Zur Berechnung wurde das lokale Dateisystem verwendet. Die Knoten tauschen in der merge-Phase die Daten via HTTP aus. Da in jedem Knoten nur eine Festplatte vorhanden ist, können nicht alle 8 Threads genutzt werden. Anderenfalls käme es in den Map- und Reduce-Phasen, obwohl jeder einzelne Thread die Buckets als Stream liest, zu random-access auf den Festplatten. Deshalb wurde die Zahl der Prozesse pro Knoten auf zwei reduziert. Eine weitere Einschränkung ergab sich durch die Funktion zur Partitionierung der Daten, die auf der Adler Prüfsumme (vgl. Deutsch und Gailly, 1996) basiert. Zur Bestimmung der Partition wird die Prüfsumme des Zustands modulo der Anzahl der Partitionen berechnet. Diese Funktion zeigt allerdings Schwächen für eine gerade Anzahl an Partitionen, weshalb die Gesamtzahl der Prozesse auf 61 reduziert wurde.

Jede Partition der 8+8+8 PDB wurde einzeln in mehreren Schritten erstellt. Um eine Partition zu erstellen waren, ca. 72 Stunden notwendig. Tabelle 5.1 zeigt die Zahl der Zustände des Patterns (1) aus Abbildung 5.1. Insgesamt enthält der Suchgraph 87.358.400.640 Zustände[1]. Die größte Suchfront besteht aus 9.617.642.082 Zuständen.

Das ist nach unserem Wissen die bisher größte jemals erstellte PDB.

5.1.2. Zehn 6+6+6+6 PDBs

Aus dem vierten Kapitel dieser Arbeit geht hervor, dass große PDBs zusammen mit einigen kleinen effizient sind. Deshalb wurden für eine große Gruppe mit der oben vorgestellten 8+8+8 PDB zusätzlich zehn 6+6+6+6 PDBs erzeugt. Eine grafische Darstellung der Partitionierungen befindet sich im Anhang (Abb. A.2), darunter auch die in Korf und Felner (2002) erwähnte Partitionierung (Abb. A.1(a)). Die restlichen Partitionierungen wurden von Hand entworfen. Jede dieser PDBs wurde mit MR-BFFS erzeugt und nach der Position des Blanks komprimiert gespeichert (vgl. Felner et al., 2007), so dass eine 6+6+6+6 PDB 486,42 MBytes im Hauptspeicher belegt.

[1] Der Suchgraph hat weniger als $\frac{25!}{(25-8-1)!}$ Zustände, da beim Erstellen von PDBs mit MR-BFFS in einer Tiefe alle Zustände mit gleichem Pattern (ohne Blank) und einem Zustand zusammengefasst werden. Die unterschiedlichen Positionen des Blanks werden in diesem Zustand in einer separaten Liste gespeichert.

Aufgrund einander überlappender Partitionen beträgt die Speichergröße der Gruppe aller zehn PDBs mit 3.782 MBytes etwas weniger als die Summe der einzelnen PDBs.

5.2. Heuristische Suche mit BFIDA*

In diesem Teil wird der Algorithmus BFIDA* und dessen Anwendung auf MapReduce genauer untersucht. Der erste Abschnitt vergleicht das Suchverfahren mit der iterativen heuristischen Tiefensuche IDA*. Die Leistungen der erzeugten PDBs werden im zweiten Abschnitt verglichen. Zuletzt wird die Skalierbarkeit von MR-BFIDA* analysiert.

5.2.1. BFIDA* vs. IDA*

Die beiden Algorithmen IDA* und BFIDA* unterscheiden sich in mehreren Merkmalen. IDA* ist nicht in der Lage, Zyklen im Suchgraphen zu erkennen, weil es keine Information über bereits besuchte Knoten speichert. Einerseits hat das den Vorteil, dass dieses Suchverfahren eine sehr geringe Speicheranforderung hat, führt zum anderen aber zu einer größeren Knotenexpansion. Zudem stoppt IDA* bei der ersten gefundenen Lösung. Da IDA* auf der Tiefensuche basiert, kann es durch Zufall sehr früh in der letzten Iteration einen Lösungspfad finden oder sehr spät auf eine Lösung treffen. Im Gegensatz dazu durchsucht BFIDA* jede Tiefe vollständig, findet daher alle Lösungen und besucht durch das Erkennen von Duplikaten wesentlich weniger Knoten. Um diesen Effekt zu verdeutlichen, wurden alle 50 Instanzen des 24er Puzzles aus Korf und Felner (2002), unter Verwendung der 6+6+6+6 PDB aus (Abb. A.1(a)) inkl. der Spiegelung um die Hauptdiagonalen, optimal gelöst. Die Probleme wurden aufsteigend nach der Zahl der Knoten sortiert, die IDA* zum Lösen benötigt, um eine Ordnung nach der Schwere der Probleme zu erhalten. Die geordneten Probleminstanzen sind in Tabelle A.3 im Anhang zu finden. Eine Auflistung aller Ergebnisse befindet sich in Tabelle 5.2. In der ersten Spalte steht die ID, mit der das Problem in Tabelle A.3 gelistet ist, die Länge des kürzesten Lösungspfads ist in der zweiten Spalte angegeben. Spalte drei nennt die Zahl der besuchten Knoten aus Korf und Felner (2002), die Korfs IDA* zum Lösen der jeweiligen Instanz benötigt. Die vierte Spalte notiert die Zahl der besuchten Knoten von BFIDA* und die Letzte die Reduktion der expandierten Knoten durch Verwendung von BFIDA*. Die Durchschnittswerte sind der letzten Zeile zu entnehmen. BFIDA* expandiert im 24er-Puzzle im Gegensatz zu IDA* durchschnittlich 5,11 mal weniger Knoten. Dieser Faktor ist abhängig von der Zahl der Zyklen im Suchgraphen. Abbildung 5.2 stellt die

ID	Tiefe	Korf IDA* (Knoten)	BF-IDA* (Knoten)	Reduktion
38	96	38.173.507	58.097.633	0,66
40	82	65.099.578	26.320.497	2,47
25	81	292.174.444	127.949.696	2,28
32	97	428.222.507	399.045.498	1,07
44	93	867.106.238	181.555.996	4,78
37	100	1.496.759.944	1.646.715.005	0,91
30	92	1.634.941.420	661.835.606	2,47
13	101	1.959.833.487	1.979.587.555	0,99
1	95	2.031.102.635	1.059.622.872	1,92
28	98	2.258.006.870	450.493.295	5,01
36	90	2.582.008.940	603.580.192	4,28
5	100	2.899.007.625	1.859.102.197	1,56
22	95	3.592.980.531	581.539.254	6,18
16	96	3.803.445.934	1.783.144.872	2,13
29	88	4.787.505.637	1.090.385.785	4,39
4	98	10.991.471.966	5.154.861.019	2,13
26	105	12.397.787.391	6.039.700.647	2,05
3	97	21.148.144.928	4.805.007.493	4,40
31	99	26.200.330.686	7.785.405.374	3,37
41	106	26.998.190.480	8.064.453.928	3,35
47	92	30.443.173.162	4.385.270.986	6,94
27	99	53.444.360.033	7.884.559.441	6,78
43	104	55.147.320.204	8.816.151.498	6,26
46	100	65.675.717.510	21.674.806.323	3,03
45	101	79.148.491.306	17.068.061.084	4,64
6	101	103.460.814.368	9.810.208.759	10,55
7	104	106.321.592.792	27.686.193.468	3,84
49	100	108.197.305.702	11.220.738.849	9,64
35	98	116.131.234.743	23.049.423.391	5,04
8	108	116.202.273.788	29.575.219.906	3,93
39	104	161.211.472.633	34.198.605.172	4,71
23	104	171.498.441.076	54.281.904.788	3,16
15	103	173.999.717.809	52.178.879.610	3,33
2	96	211.884.984.525	40.161.477.151	5,28
19	106	218.284.544.233	22.761.173.348	9,59
42	108	245.852.754.920	37.492.323.962	6,56
20	92	312.016.177.684	20.689.215.063	15,08
24	107	357.290.691.483	38.272.741.957	9,34
17	109	367.150.048.758	143.972.316.747	2,55
34	102	481.039.271.661	59.225.710.222	8,12
48	107	555.085.543.507	58.365.224.981	9,51
12	109	624.413.663.951	76.476.143.041	8,16
21	103	724.024.589.335	98.083.647.769	7,38
18	110	987.725.030.433	126.470.260.027	7,81
33	106	1.062.250.612.558	134.103.676.989	7,92
14	111	1.283.051.362.385	312.885.453.572	4,10
10	114	1.519.052.821.943	525.907.193.133	2,89
11	106	1.654.042.891.186	309.253.017.124	5,35
9	113	1.818.005.616.606	132.599.245.368	13,71
50	113	4.156.099.168.506	1.067.321.687.213	3,89
∅	**100,78**	**360.892.479.670**	**71.004.578.707**	**5,11**

Tabelle 5.2.: 50 Instanzen des 24er-Puzzles aus Korf und Felner (2002) mit der 6+6+6+6 PDB (Abb. A.1(a)) inkl. Spiegelung um die Hauptdiagonale gelöst.

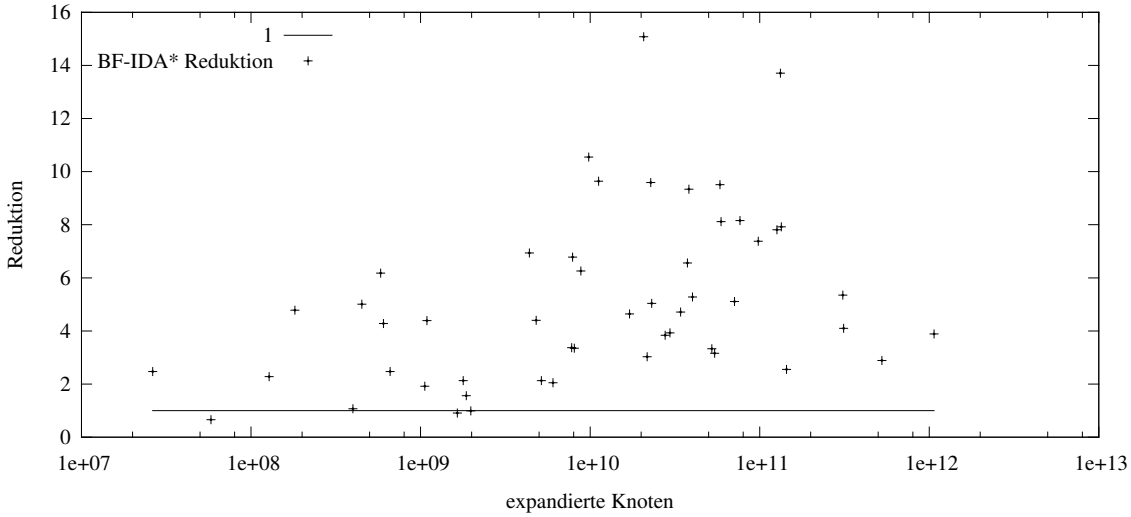


Abbildung 5.2.: Reduktion von BFIDA* gegenüber IDA* beim Lösen der 50 Zufallsinstanzen des 24er-Puzzles aus Korf und Felner (2002) unter Verwendung der besten 6+6+6+6 PDB mit Spiegelung um die Hauptdiagonale.

Ergebnisse graphisch dar, wobei die Reduktion von BFIDA* der Zahl der expandierten Knoten mit diesem Suchverfahren gegenüber gestellt ist. BFIDA* profitiert stärker bei schwereren Problemen, weil IDA* in diesen mehr Zyklen durchläuft. Das unregelmäßige Muster ist dem Umstand geschuldet, dass IDA* bei der ersten Lösung stoppt, während BFIDA* alle Lösungen findet.

5.2.2. Leistung der PDBs im Vergleich

Dieser Abschnitt vergleicht die Leistungen der oben erzeugten PDBs unter Anwendung von MR-BFIDA*. Die Bewertung soll mit Hilfe des optimalen Lösens von 50 zufälligen Problemen aus Korf und Felner (2002) erfolgen.

Im ersten Versuch wird die Gruppe der zehn 6+6+6+6 PDBs untersucht, wobei die ersten zehn Instanzen (Tab. A.3) mit allen 1023 ($\sum_{k=1}^{10} \binom{10}{k}$) möglichen Teilmengen mit 1 bis 10 Elementen gelöst werden. Die Spiegelung um die Hauptdiagonale wurde dabei berücksichtigt. Das Ergebnis ist graphisch in Abbildung 5.3 festgehalten, in der beide Achsen logarithmisch skaliert sind. Für jede Gruppe ist die Summe der expandierten Knoten zur Speicherbelegung der Gruppe dargestellt und es ergibt sich ein ähnliches Bild wie zu den in Kapitel 4 vorgestellten Ergebnissen (Abb. 4.4). Demnach nimmt mit steigender Speicherbelegung die Zahl der expandierten Knoten ab, dieser Effekt wird

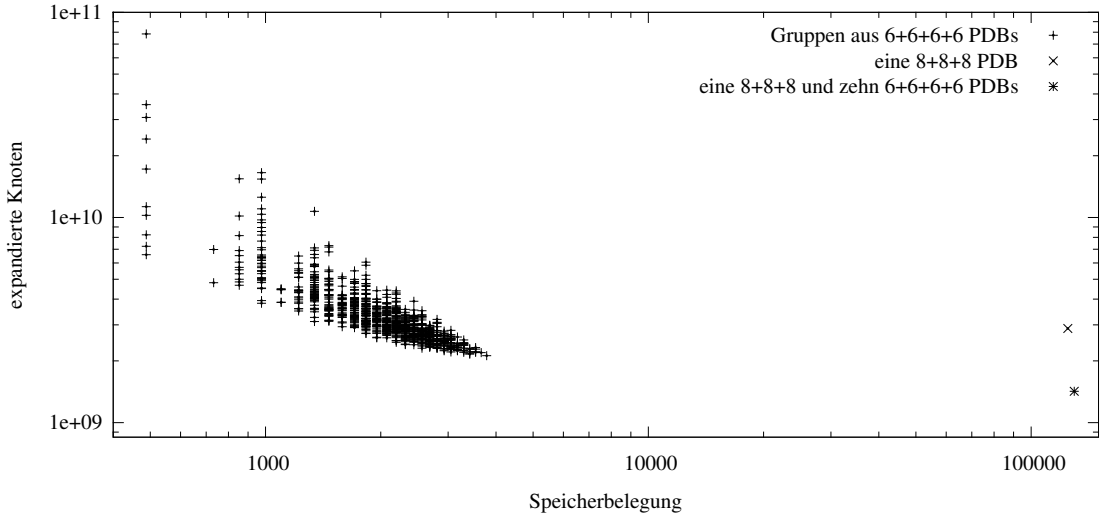


Abbildung 5.3.: Summe der expandierten Knoten (Y-Achse, log. Skala) zur Speichergröße der Heuristik (X-Achse, log. Skala) für ausgewählte PDB Gruppen mit Spiegelung um die Hauptdiagonale im 24er-Puzzle. Gelöst wurden die ersten zehn Probleminstanzen aus Tabelle A.3.

allerdings zunehmend schwächer. Am rechten Rand der Abbildung sind zwei weitere Datenpunkte erkennbar, welche die Leistung der 8+8+8 PDB sowie die Kombination der zehn 6+6+6+6 und der 8+8+8 PDB zu einer Gruppe zeigen. Während die einzelne 8+8+8 PDB ein besseres Ergebnis erzeugt als die beste einzelne 6+6+6+6 PDB, kann bereits eine Gruppe von vier 6+6+6+6 PDBs diese Leistung erreichen. Mit einer Gruppe bestehend aus der 8+8+8 und den zehn 6+6+6+6 PDBs kann jedoch die Zahl der Knoten weiter reduziert werden.

Zur Erweiterung des Versuchs wurden die ersten 45 Probleminstanzen aus Tabelle A.3 mit der besten 6+6+6+6 PDB (Abb. A.1(a)), der erstellten 8+8+8 PDB, dem Maximum der zehn 6+6+6+6 PDBs, sowie der Gruppe bestehend aus den zehn 6+6+6+6 PDBs und der 8+8+8 PDB gelöst. Für jede PDB wurde zusätzlich der Heuristikwert durch Spiegelung an der Hauptdiagonalen berechnet, so dass insgesamt über 22 Heuristikwerte maximiert wurde. Aufgrund begrenzter Kapazität im Hauptspeicher konnten die letzten fünf Probleme nicht gelöst werden. In Tabelle 5.3 sind die Ergebnisse aufgelistet. Die erste Spalte gibt die ID an, unter der das Problem in Tab. A.3 gelistet ist, die Länge des kürzesten Lösungspfads ist in Spalte zwei angegeben. Die dritte Spalte notiert die Anzahl der Knoten, die benötigt werden, das Problem mit Hilfe der besten 6+6+6+6

PDB zu lösen und analog sind in der vierten Spalte die Ergebnisse für die 8+8+8 PDB aufgelistet. Die fünfte Spalte notiert die Reduktion R der expandierten Knoten durch die 8+8+8 PDB. Die Ergebnisse für die Gruppe der zehn 6+6+6+6 PDBs sind den Spalten sechs und sieben zu entnehmen. Die letzten beiden Spalten beinhalten das Resultat der Kombination aus den zehn 6+6+6+6 PDBs sowie der 8+8+8 PDB. Die Durchschnittswerte der jeweiligen Spalte sind in der letzten Zeile angegeben.

Durch den Einsatz der 8+8+8 PDB kann die Zahl der expandierten Knoten im Vergleich zur Referenzheuristik um durchschnittlich 2,85 reduziert werden. Die Reduktion schwankt dabei zwischen 8,46 und 0,53. Bei vier der 45 Problemen ist die 8+8+8 PDB schlechter als die 6+6+6+6 PDB. Die Leistung der Gruppe mit zehn 6+6+6+6 PDBs ist deutlich besser, da eine durchschnittliche Reduktion von 4,20 erreicht wird. Bei dieser Gruppe schwankt die Reduktion zwischen 11,25 und 2,46. Für die Kombination aller PDBs variiert die Reduktion von mindestens 3,33 bis höchstens 24,08, während sie durchschnittlich 7,86 beträgt. Durch die Kombination der großen PDB mit einer Menge kleiner kann die Leistung weiter verbessert werden. Im Vergleich zu BFIDA* mit der besten 6+6+6+6 PDB wird die Knotenexpansion um Faktor 7,86 und verglichen mit IDA* sogar um Faktor 34,83 reduziert.

5.2.3. Skalierbarkeit von MR-BFIDA*

Nachfolgend wird die Skalierbarkeit von MR-BFIDA* in der in-memory Variante untersucht. Grundsätzlich sind die beiden Betriebsarten shared-memory mit OpenMP und distributed shared-memory mit MPI zu unterscheiden, da die Parallelisierung beider Arten sehr unterschiedlich implementiert wird. Zunächst wird die theoretische Grundlage für die Messung der Skalierbarkeit erläutert.

Die Skalierbarkeit eines Algorithmus wird mit dem *Speedup S(P)* in Abhängigkeit zur Zahl der eingesetzten Prozessoren P gemessen. Der Speedup ist definiert als der Quotient der Ausführungszeit des schnellsten sequentiellen Algorithmus zur Ausführungszeit mit P Prozessoren.

$$S(P) = \frac{T(1)}{T(N)}$$

Als schnellster sequentieller Algorithmus zum Lösen des Problems des kürzesten Pfades gilt IDA*. Ein Vergleich mit diesem Algorithmus ist mitunter problematisch, da dieser zum einen nur die erste optimale Lösung bestimmt und danach terminiert, während BFIDA* alle optimalen Lösungen findet. Andererseits ist in der vorliegenden Implemen-

ID	Tiefe	[1] 6+6+6+6	[2] 8+8+8	R	[3] 10x6+6+6+6	R	[2] & [3]	R
38	96	58.097.633	9.577.883	6,07	13.289.381	4,37	5.524.442	10,52
40	82	26.320.497	49.291.000	0,53	10.134.216	2,60	7.904.477	3,33
25	81	127.949.696	118.780.897	1,08	37.690.831	3,39	26.362.335	4,85
32	97	399.045.498	281.515.091	1,42	167.956.756	2,38	98.785.271	4,04
44	93	181.555.996	37.853.812	4,80	40.797.152	4,45	15.486.572	11,72
37	100	1.646.715.005	628.890.120	2,62	654.232.285	2,52	348.866.269	4,72
30	92	661.835.606	256.431.250	2,58	176.840.132	3,74	75.318.276	8,79
13	101	1.979.587.555	1.181.771.575	1,68	762.110.447	2,60	396.551.893	4,99
1	95	1.059.622.872	199.198.406	5,32	157.426.679	6,73	73.300.296	14,46
28	98	450.493.295	114.571.662	3,93	102.432.224	4,40	45.934.516	9,81
36	90	603.580.192	408.261.989	1,48	210.388.791	2,87	124.378.030	4,85
5	100	1.859.102.197	3.125.977.623	0,59	389.166.721	4,78	338.063.069	5,50
22	95	581.539.254	82.503.279	7,05	182.750.369	3,18	44.032.593	13,21
16	96	1.783.144.872	1.729.554.795	1,03	543.458.449	3,28	419.061.531	4,26
29	88	1.090.385.785	128.886.129	8,46	123.498.200	8,83	45.286.680	24,08
4	98	5.154.861.019	1.361.290.863	3,79	1.077.714.057	4,78	483.250.833	10,67
26	105	6.039.700.647	4.993.857.550	1,21	2.215.844.828	2,73	1.426.275.074	4,23
3	97	4.805.007.493	5.699.072.723	0,84	1.426.715.024	3,37	1.079.945.735	4,45
31	99	7.785.405.374	3.653.831.114	2,13	3.609.060.880	2,16	1.925.753.655	4,04
41	106	8.064.453.928	1.737.010.534	4,64	1.926.278.571	4,19	813.759.866	9,91
47	92	4.385.270.986	3.825.636.827	1,15	1.454.541.857	3,01	1.041.538.566	4,21
27	99	7.884.559.441	1.415.859.414	5,57	700.669.053	11,25	431.438.515	18,28
43	104	8.816.151.498	4.378.714.353	2,01	2.367.037.484	3,72	1.502.646.189	5,87
46	100	21.674.806.323	9.872.851.915	2,20	3.557.844.707	6,09	2.258.356.818	9,60
45	101	17.068.061.084	5.614.562.048	3,04	5.092.440.386	3,35	2.053.582.750	8,31
6	101	9.810.208.759	2.397.434.227	4,09	1.425.769.239	6,88	787.876.397	12,45
7	104	27.686.193.468	26.781.188.637	1,03	8.154.816.631	3,40	5.458.732.047	5,07
49	100	11.220.738.849	5.526.627.744	2,03	2.701.413.795	4,15	1.583.402.497	7,09
35	98	23.049.423.391	8.584.994.059	2,68	3.326.410.623	6,93	2.423.508.016	9,51
8	108	29.575.219.906	4.318.849.565	6,85	8.142.420.338	3,63	2.961.519.065	9,99
39	104	34.198.605.172	22.810.919.845	1,50	6.546.789.727	5,22	5.118.055.220	6,68
23	104	54.281.904.788	36.611.741.317	1,48	13.827.460.757	3,93	10.638.081.510	5,10
15	103	52.178.879.610	26.951.022.561	1,94	12.843.934.629	4,06	8.962.516.108	5,82
2	96	40.161.477.151	29.318.072.174	1,37	16.326.392.138	2,46	10.058.017.530	3,99
19	106	22.761.173.348	6.759.987.121	3,37	5.557.331.495	4,10	3.042.217.781	7,48
42	108	37.492.323.962	9.339.335.844	4,01	12.209.190.559	3,07	5.058.488.637	7,41
20	92	20.689.215.063	9.014.702.404	2,30	3.378.087.267	6,12	2.411.282.328	8,58
24	107	38.272.741.957	25.802.863.114	1,48	10.941.544.654	3,50	7.010.438.590	5,46
17	109	143.972.316.747	49.516.974.145	2,91	29.452.088.936	4,89	17.862.300.871	8,06
34	102	59.225.710.222	49.923.377.951	1,19	12.038.955.444	4,92	9.195.325.321	6,44
48	107	58.365.224.981	99.614.525.233	0,59	16.999.755.500	3,43	13.407.677.441	4,35
12	109	76.476.143.041	43.132.155.298	1,77	25.638.902.721	2,98	13.255.427.626	5,77
21	103	98.083.647.769	25.411.173.479	3,86	25.088.589.852	3,91	11.165.798.532	8,78
18	110	126.470.260.027	18.375.847.744	6,88	35.622.777.161	3,55	10.756.778.817	11,76
33	106	134.103.676.989	77.163.409.262	1,74	43.594.870.726	3,08	25.969.921.576	5,16
Ø	**99,6**	**26.716.940.865**	**13.961.354.546**	**2,85**	**7.129.284.926**	**4,20**	**4.049.083.781**	**7,86**

Tabelle 5.3.: Die ersten 45 Probleminstanzen aus Tab. A.3 mit BFIDA* unter Verwendung verschiedener Heuristiken gelöst. Die Spiegelung um die Hauptdiagonale wurde berücksichtigt.

P	$T(P)$	$S(P)$
1	23044,3	1,00
2	11013,3	2,09
4	5559,2	4,15
7	3200,3	7,20
17	1482,1	15,55
31	1134,7	20,31

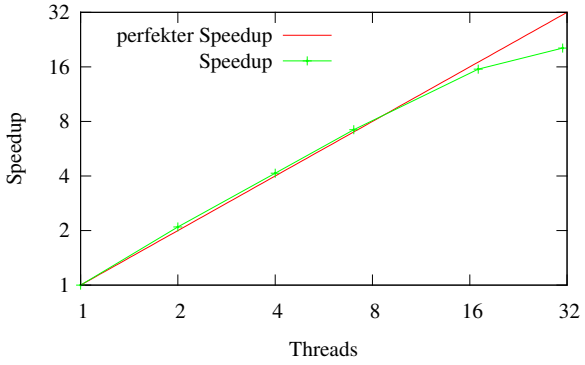


Abbildung 5.4.: Skalierbarkeit von MR-BFIDA* mit OpenMP

tierung von BFIDA* keine Routine zur Wiederherstellung des kürzesten Pfades realisiert. Somit sind die Ergebnismengen von IDA* und MR-BFIDA* unterschiedlich und es bleibt als Vergleich die sequentielle Ausführung von MR-BFIDA*.

OpenMP

Um die Skalierung für MR-BFIDA* in der OpenMP-Variante zu testen, wurden im 24er Puzzle die ersten 10 Probleme aus Tabelle A.3 gelöst. Das Testsystem ist ein shared-memory System, ausgestattet mit acht AMD Quad-Core Opteron 8358 SE (2,4 GHz) und 256 GByte Hauptspeicher. Die beschriebene Menge an Probleminstanzen wird mit MR-BFIDA* unter Verwendung von 1, 2, 4, 7, 17 und 31 Kernen gelöst. Die ungeraden Zahlen sind, wie beim Erstellen der 8+8+8 PDB in Kapitel 5.1.1, der Funktion zur Partitionierung der Daten geschuldet.

Die Ergebnisse sind in Abbildung 5.4 tabellarisch und grafisch dargestellt. Die erste Spalte der Tabelle gibt die Anzahl der Kerne an. Die Zeit, die zum Lösen aller Probleminstanzen nötig ist, steht in Spalte zwei. Die dritte Spalte enthält den Speedup relativ zu MR-BFIDA* mit einem Kern.

In der OpenMP-Variante skaliert MR-BFIDA* für bis zu 7 Kerne linear. Für 17 und 31 Kerne nimmt die Effizient der Parallelisierung wegen der begrenzten Hauptspeicherbandbreite ab. Die Daten werden schneller verarbeitet als sie zwischen den CPUs und dem Hauptspeicher transferiert werden können. Eine analoge Messung auf einem 8-Kerne System bestätigt diese These. Für große Systeme wie beispielsweise SGI UV wird erwartet, dass auch dort die Hauptspeicherbandbreite die Skalierung beschränken wird.

P	Prob.	$V(P)$	$S(P)$
1	30	5.16e+05	1,00
7	3	3.33e+06	6,47
31	27	1.52e+07	29,46
63	19	3.05e+07	59,11
127	2	5.60e+07	108,53
253	21	1.10e+08	213,18
511	10	1.96e+08	379,84
2039	50	4.42e+08	856,59

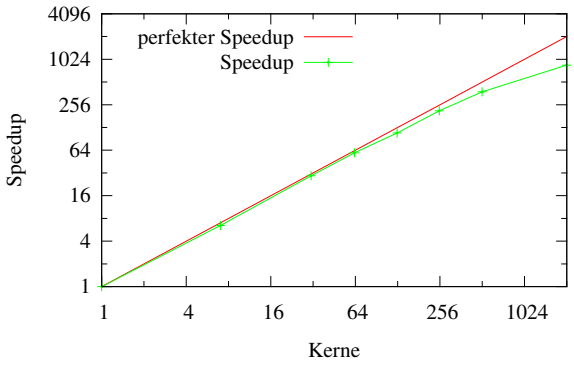


Abbildung 5.5.: Skalierbarkeit von MR-BFIDA* mit MPI

MPI

Weil die Effizienz der MPI-Variante des Algorithmus von einer angemessenen Problemstellung abhängig ist, wurde für deren Evaluierung ein anderes Verfahren gewählt. Je nach Anzahl der eingesetzten Kerne wird eine adäquate Probleminstanz gelöst. Obwohl das Hinzunehmen eines weiteren Knotens die Verarbeitungsgeschwindigkeit des Gesamtsystems erhöht und damit die Ausführungszeit verringert, treten erhöhte Kosten für Kommunikation und Synchronisation auf. Bei einem leichten Problem auf einem großen System wird die Ausführungszeit von diesen Komponenten dominiert, während die Zeit der eigentlichen Berechnung in den Hintergrund rückt. Ein schweres Problem auf einem kleinen System kann nicht gelöst werden, da hierfür der Hauptspeicher nicht ausreicht. Um die verschiedenen Programmläufe miteinander vergleichen zu können, wird die Geschwindigkeit $V(P)$ angegeben, welche die Zahl der expandierten Knoten in das Verhältnis zur gesamten Ausführungszeit (*wall time*) setzt.

Getestet wird die MPI-Variante auf einem Cluster mit je zwei Quad-Core Intel Xeon X5570 (2,93 GHz) und 48 GByte Hauptspeicher. Auf diesem System wird mit 1, 7, 31, 63, 127, 253, 511 und 2039 Kernen gerechnet. Die Ergebnisse sind in Abbildung 5.5 tabellarisch und grafisch dargestellt. Die erste Spalte der Tabelle gibt die Zahl der Kerne an, die Nummer der entsprechenden Probleminstanz (vgl. Tab A.3) steht in Spalte zwei. Die dritte Spalte nennt die erreichte Geschwindigkeit V. Die letzte Spalte notiert den Speedup.

In der MPI-Variante skaliert MR-BFIDA* bis 63 Kerne nahezu perfekt und ab 511 Kernen ist ein deutlicher Einbruch zu verzeichnen, welcher im Folgenden genauer un-

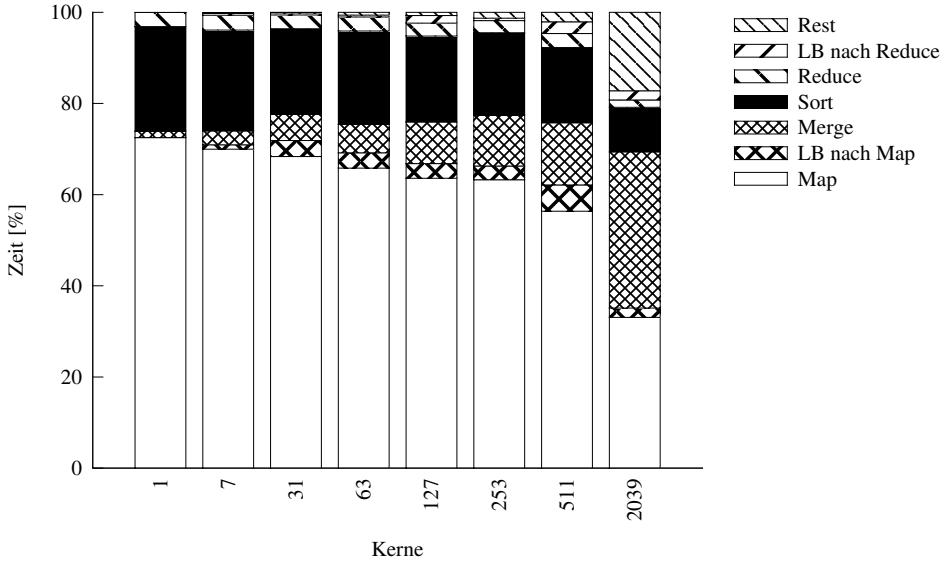


Abbildung 5.6.: Ausführungsprofil von MR-BFIDA* für verschiedene Systemgrößen.

tersucht wird. An dieser Stelle sei an die Beschreibung des MR-Search Frameworks (Kap. 3.5) erinnert. Darin ist beschrieben, dass die in-memory MPI-Implementation nach der Map-Phase die Daten mittels dem MPI-Befehl `MPI_Alltoallv` zwischen den Knoten verteilt. Da vor dem eigentlichen Datentransfer ein Verbindungsaufbau zu *jedem* Prozess durchgeführt werden muss, steigt der Aufwand linear mit jedem hinzugefügten Prozess (vgl. Balaji et al., 2009; Hoefler et al., 2009), weshalb die Merge-Phase in den Fokus der Betrachtung um die mangelnde Skalierbarkeit rückt. Abbildung 5.6 zeigt ein Ausführungsprofil für MR-BFIDA*. Jeder Balken repräsentiert einen Programmlauf mit entsprechender Anzahl an Kernen (X-Achse), wobei die prozentualen Anteile der verschiedenen Programmphasen Map, Lastbalancierung (LB) nach Map, Merge, Sortieren, Reduce, LB nach Reduce und Rest abgetragen sind. Die Lastbalancierung findet nach der Map- und der Reduce-Phase statt, wobei alle Prozesse warten müssen, bis der Letzte die entsprechende Aufgabe beendet hat. Erst dann können die Prozesse mit dem nächsten Schritt fortsetzen. Es ist deutlich zu sehen, dass die merge-Phase die Gesamtzeit immer stärker dominiert. Der Algorithmus verbringt bei 31 Kernen nur knapp 6% der Ausführungszeit in dieser Phase. Bei 511 Kernen sind es schon knapp 14% und bei 2039 CPUs ganze 34%. Eine genaue Analyse der Merge-Phase erlauben die Abbildungen 5.7 und 5.8, in denen die durchschnittliche Zeit und das durchschnittliche Datenvolumen pro Knoten jeder Merge-Phasen dargestellt sind. In Abbildung 5.7 wird das Problem

Nummer 10 aus Korf und Felner (2002) mit 511 Kernen gelöst, in Abbildung 5.8 das Problem Nummer 50 mit 2039 Kernen. Im Fall von 511 Kernen steigt die durchschnittliche Zeit pro Merge-Phase nur an, wenn nennenswerte Datenmengen übertragen werden. Dagegen benötigt jede Merge-Phase mit 2039 Prozessoren mindestens 0,7 Sekunden.

5.2.4. MR-BFIDA* mit PDBs von Festplatte

In Kapitel 3.4.2 wurde ein Schema vorgestellt, mit dem sehr große PDBs während der Map-Phase direkt von der Festplatte gelesen werden können. Um dieses Schema auf die Anwendbarkeit für das Lösen von Probleminstanzen des 24er-Puzzles zu untersuchen, müssen die auftretenden Größen der Suchfronten während der Suche betrachtet werden. Für jeden Schwellwert $thresh$ werden alle Tiefen $1 \leq g \leq thresh$ durchsucht. Die Suchfronten wachsen mit diesem Threshold exponentiell an, wie die Darstellung des übertragenen Datenvolumens in Abbildung 5.8 andeutet. Jede Erhebung des Datenvolumens in der Abbildung entspricht der Suche mit einem vorgegebenen Threshold. Jede Tiefe g jedes Thresholds entspricht einem MapReduce-Schritt. Aus der Abbildung geht hervor, dass die wenigsten MapReduce-Schritte datenintensiv sind. Es werden beispielsweise nur in 102 von 764 (13%) der Merge-Phasen 10 GBytes und mehr transferiert. Für den effizienten Einsatz von Festplatten sind jedoch MapReduce-Schritte mit einer großen Menge an Daten notwendig, wie es bei dem Erzeugen der PDBs der Fall ist. Da die vorliegende Implementation des MapReduce-Frameworks keinen dynamischen Wechsel von in-memory zu out-of-core Berechnungen erlaubt, ist ein effizientes out-of-core BF-IDA* (zur Zeit) nicht möglich.

Weiterhin ist zu beachten, dass sich das in diesem Abschnitt vorgestellte Schema nicht für die Verwendung von Gruppen von PDBs geeignet ist, weil die Erzeugung der Pattern für jede Partition erfolgen muss. Beim Einsatz von n PDBs wird die Suchfront im Map-Schritt zum Aufteilen der Position in die Pattern ver-n-facht.

5.3. Zusammenfassung

In diesem Kapitel wurden die vorgestellten Algorithmen MR-BFFS und MR-BFIDA* auf Probleme des 24er-Puzzles angewandt. Um die Leistungsfähigkeit von MR-BFFS zu demonstrieren, wurde eine vollständige 8+8+8 PDB erzeugt. Das ist nach unserem Wissen die bisher größte jemals erstellte PDB. Da eine einzelne PDB ineffizient ist, wurden außerdem zehn 6+6+6+6 PDBs erstellt.

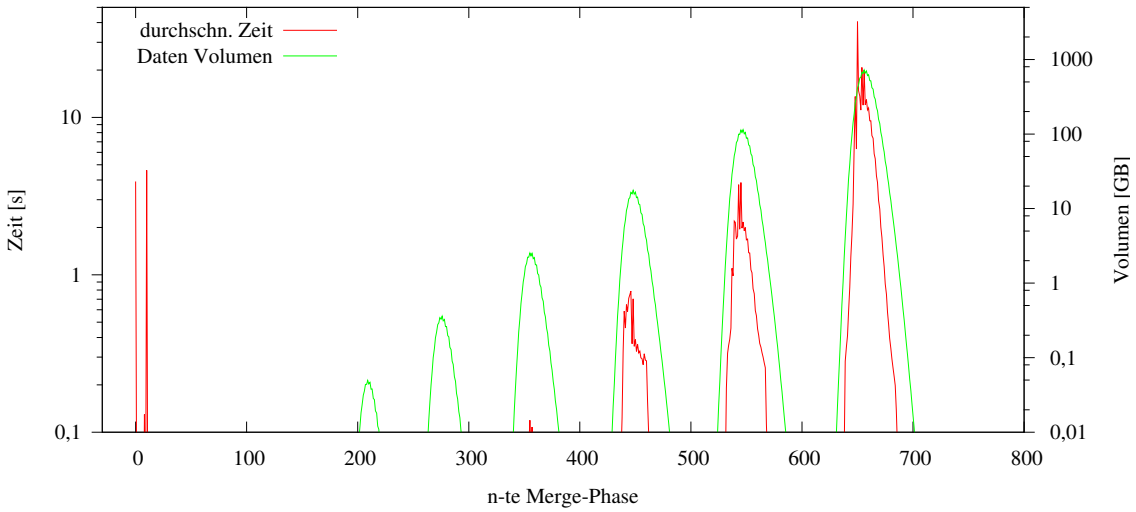


Abbildung 5.7.: Zeit und Datenvolumen für die n-te Merge-Phase beim Lösen des Problems #9 mit 511 Prozessoren

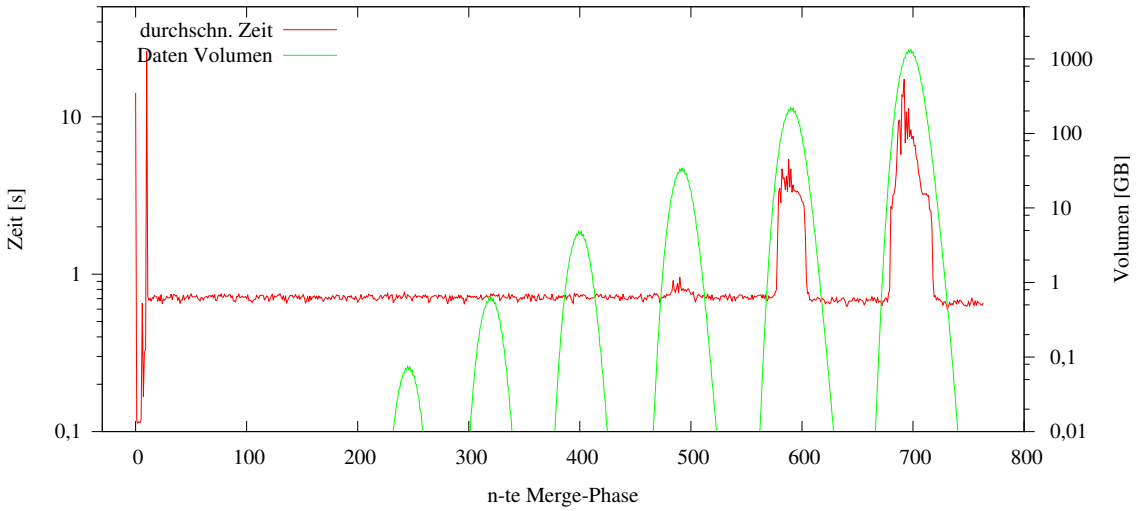


Abbildung 5.8.: Zeit und Datenvolumen für die n-te Merge-Phase beim Lösen des Problems #50 mit 2039 Prozessoren

Mit diesen PDBs wurden zufällige Instanzen des 24er-Puzzles mit MR-BFIDA* gelöst. Dabei wurde zunächst festgestellt, dass BFIDA* im Vergleich zu IDA* 5,11 mal weniger Knoten expandiert, was vor allem an Fähigkeit von BFIDA* liegt, Zyklen zu erkennen. Anschließend wurde die Leistung der erzeugten PDBs analysiert. Hier konnten Ergebnisse aus Kapitel 4 für das 24er-Puzzle bestätigt werden. Die Suche mit der einzelnen 8+8+8 PDB expandiert mehr Knoten als eine Gruppe mit zehn 6+6+6+6 PDBs. Wird die 8+8+8 PDB zur Gruppe hinzugenommen, kann die Zahl der Knoten beinahe halbiert werden.

Ferner wurde die Skalierbarkeit von MR-BFIDA* in der in-memory Variante analysiert. Sowohl mit OpenMP als auch mit MPI skaliert MR-BFIDA* für weite Teile nahezu perfekt. Die Skalierbarkeit der ersteren wird durch eine begrenzte Hauptspeicherbandbreite gemindert, während bei der MPI-Variante die Kosten der Kommunikation stark ansteigen. Für MR-BFIDA* mit MPI konnte bei 2039 Kernen eine Speedup von 856,59 nachgewiesen werden.

Eine abschließende Analyse von MR-BFIDA* hinsichtlich der Verwendbarkeit von Festplatten im 24er-Puzzle hat gezeigt, dass ein dynamischer Wechsel zwischen in-memory und out-of-core Berechnungen benötigt wird.

6. Fazit

Wie eingangs erläutert wird, schaffen Reinefeld und Schütt (2009) mit dem Algorithmus MR-BFFS erstmals die Möglichkeit, eine massiv-parallele out-of-core Suche in beliebigen Domänen durchzuführen. In dieser Arbeit wurde MR-BFFS um die Möglichkeit erweitert, sehr große PDBs erstellen zu können. Dazu musste zunächst eine Anpassung für Suchgraphen mit ungeraden Zyklen vorgenommen werden. Durch eine geeignete Zustandsrepräsentation und Kostenfunktion konnte daraufhin die Breitensuche für das Erstellen von sehr großen PDBs verwendet werden. So entstand im Laufe dieser Arbeit die erste vollständige 8+8+8 PDB.

Für die heuristische Suche mit sehr großen PDBs wurde ein Ablaufschema für MR-BFIDA* entwickelt, mit dem die Heuristikwerte direkt von der Festplatte gelesen werden können. Dazu wird erst in einem vorbereitenden Map-Schritt der Zustand in entsprechende Pattern umgewandelt. Der darauf folgende Reduce-Schritt kann dann diese Pattern mit den entsprechenden Heuristikwerten kombinieren. Auf diese Weise ist eine massiv-parallele Suche mit sehr großen PDBs möglich.

Um die Algorithmen auf MapReduce anzuwenden, wurde im Rahmen dieser Arbeit das generische Framework MR-Search entwickelt. Es beinhaltet unterschiedliche Strategien zur Parallelisierung und zur Datenhaltung. MR-Search kann einfach auf weitere Domänen angepasst werden. Beide Algorithmen skalieren nahezu perfekt. Der Speedup wird, bedingt durch die Implementation, bei großen Systemen gedämpft.

In einer Untersuchung zum Verhalten von PDBs in einer Gruppe wurde der Einfluss der Faktoren Gruppengröße und Speicherbelegung auf die Leistung der Gruppe studiert. Beide Faktoren korrelieren positiv mit der Leistung. Diese Ergebnisse können durch die Resultate im 24er-Puzzle bestätigt werden. Des Weiteren stellte die Untersuchung heraus, dass der Einsatz von großen PDBs nur dann sinnvoll ist, wenn der zur Verfügung stehende Hauptspeicher ausreicht, eine Gruppe zusammen mit einigen kleineren PDBs zu bilden.

In der abschließenden Anwendung der Algorithmen auf der 24er-Puzzle wurden zunächst die Suchalgorithmen IDA* mit BFIDA* verglichen. Durch das Erkennen von Zyklen im

Suchgraphen besucht BFIDA* im 24er Puzzle im Durchschnitt 5,11 mal weniger Knoten, als IDA*. Die erstellten PDBs können die Zahl der von BFIDA* besuchten Knoten weiter reduzieren. Eine Gruppe aus zehn 6+6+6+6 und einer 8+8+8 PDB erreicht im Mittel eine Reduktion von 7,68 gegenüber BFIDA* mit einer 6+6+6+6. Der Einsatz von Festplatten ist mit MR-BFIDA* in der derzeitigen Implementation nicht möglich, da im 24er-Puzzle zu wenige MapReduce-Schritte datenintensiv sind und somit die Latenz des Festplattenzugriffs überwiegt.

Ausblick

Die Arbeit bietet an verschiedenen Stellen Anküpfungspunkte für eine weiterführende Forschung. Zunächst sollte eine stärker anwendungsorientierte Domäne als das Schiebepuzzle betrachtet werden. So könnte beispielsweise mit Multiple Sequence Alignment eine Anwendung aus der Bioinformatik implementiert werden. Dabei werden mehrere DNA-Stränge durch Einfügen von Leerstellen aneinander angeglichen. Diese Aufgabe kann durch eine heuristische Suche gelöst werden.

Ebenso kann die Untersuchung der Gruppen auf weitere Domänen ausgeweitet werden, um dort die herausgestellten Ergebnisse zu verifizieren. Um Verallgemeinerungen für andere Domänen zu ermöglichen, kann der Versuch beispielsweise für Rubik's Cube, Türme von Hanoi oder dem Pancake Puzzle wiederholt werden.

Nicht zuletzt bleiben in der Implementation von MR-Search einige offene Punkte. Wie schon in der Evaluation in Kapitel 5 erwähnt, ist für den effizienten Einsatz von Festplatten bei der Suche mit MR-BFIDA* ein dynamischer Wechsel von in-memory zu out-of-core Berechnungen unerlässlich. Des Weiteren bedarf es im Fall von in-memory MPI einer Verbesserung der merge-Phase. Hier ist eine lastabhängige Implementierung erstrebenswert, um das Problem der mangelnden Skalierbarkeit des Befehls `MPI_Alltoall` zu umgehen.

Begriffsindex

Bestensuche, 18
Bestensuchverfahren, 18
Breadth-First Frontier Search, 19
Breitensuche, 15

compact-Mapping, 23

dual-additive, 41

effizienter Rand, 49

Frontier Search, 19

Gruppe, 42
Gruppengröße, 42

Heuristik, 17
  effizient, 43
  effizienter als, 43
  monotone, 18
  perfekte, 18
  zulässige, 18
  zuverlässige, 43

Instance Dependent PDBs, 24
iterative-deepening A*, 19

Knoten
  erzeugen, 14
  expandieren, 14
  Tiefe, 14

Linear Conflicts, 21

Manhattan Distanz, 20

Nachfolger, 13

Operatoren, 13

Pattern, 22
  k-Pattern, 22
Pattern Datenbanken, 21
  additive, 23
Patternraum, 22
Patternsteine, 22
primäre Suche, 24
Problem, 13
  Lösung, 14
Problemraum, 13

Queue
  FIFO, 15
  LIFO, 16

Schiebepuzzle, 14
  Blank, 14
Sekundärsuche, 24
sparse-Mapping, 23
Speicherbelegung, 43
Stack, 16
Suche
  heuristische, 18

    informierte, 18
    uninformierte, 17
    zielgerichtete, 14
Suchfront, 20
Suchgraph, 14
Suchverfahren
    optimales, 16
    vollständiges, 16

Threshold, 17
Tiefensuche, 16

umkehrbar, 13

Vorgänger, 13

Zielpattern, 22
Zustandsraum, 13

Literaturverzeichnis

Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E. L., Thakur, R. und Träff, J. L. (2009): *MPI on a million processors*. In: Ropo, M., Westerholm, J. und Dongarra, J. (Hrsg.): Recent Advances in Parallel Virtual Machine and Message Passing Interface. Lecture Notes in Computer Science 5759. New York: Springer. S. 20-30.

Culberson, J. C. und Schaeffer, J. (1998): *Pattern databases*. Computational Intelligence 14(3). S. 318-334.

Dean, J. und Ghemawat, S. (2004): *MapReduce. Simplified data processing on large clusters*. In: Zucker, J.-D. undSaitta, L. (Hrsg.): Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04). San Francisco: USENIX Association. S. 137-150.

Deutsch, L. P. und Gailly, J.-L. (1996): *ZLIB Compressed Data Format Specification version 3.3*. RFC 1950.

Felner, A. und Adler, A. (2005): *Solving the 24 Puzzle with Instance Dependent Pattern Databases*. In: Abstraction, Reformulation and Approximation. Lecture Notes in Computer Science 3607. New York: Springer. S. 248-260.

Felner, A., Korf, R. E. und Hanan, S. (2004): *Additive Pattern Database Heuristics*. Journal of Artificial Intelligence Research 22. S. 279-318.

Felner, A., Korf, R. E., Meshulam, R. und Holte, R. (2007): *Compressed Pattern Databases*. Journal of Artificial Intelligence Research 30. S. 213–247.

Hansson, O., Mayer, A. und Yung, M. (1992): *Criticizing Solutions to Relaxed Models Yields Powerful Admissible Heuristics*. Information Sciences 63(3). S. 207-227.

Hart, P. E., Nilsson, N. J. und Raphael, B. (1968): *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. IEEE Transactions on Systems Science and Cybernetics 4(2). S. 100-107.

Hernádvölgyi, I. T. und Holte, R. C. (2004): *Steps Towards The Automatic Creation of Search Heuristics*. Technical Report TR04-02. Edmonton: University of Alberta.

Hoefler, T., Lumsdaine, A. und Dongarra, J. (2009): *Towards Efficient MapReduce Using MPI*. In: Ropo, M., Westerholm, J. und Dongarra, J. (Hrsg.): Recent Advances in Parallel Virtual Machine and Message Passing Interface. Lecture Notes in Computer Science 5759. New York: Springer. S. 240-249.

Holte, R. C. und Hernádvölgyi, I. T. (1999), *A Space-Time Tradeoff for Memory-Based Heuristics*. In AAAI/IAAI. Cambridge: MIT Press. S. 704-709.

Holte, R. C., Newton, J., Felner, A., Meshulam, R. und Furcy, D. (2004): *Multiple Pattern Databases*. In: Zilberstein S., Koehler J. und Koenig S. (Hrsg.): Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004). Whistler: AAAI. S. 122-131.

Hupfeld, F., Cortes, T., Kolbeck, B., Stender, J., Focht, E., Hess, M., Malo, J., Martí, J. und Cesario, E. (2008): *The XtreemFS architecture - a case for object-based file systems in Grids*. Concurrency and Computation: Practice and Experience 20(17). S. 2049-2060.

Johnson, W. und Story, W. (1879): *Notes on the "15"puzzle*. American Journal of Mathematics 2. S. 397-404.

Korf, R. E. (1985*a*): *Depth-First Iterative-Deepening. An Optimal Admissible Tree Search*. Artificial Intelligence 27. S. 97-109.

Korf, R. E. (1985*b*): *Macro-operators: a weak method for learning*. Artificial Intelligence 26. S. 35-77.

Korf, R. E. (1997): *Finding Optimal Solutions to Rubik's Cube Using Pattern Databases*. In AAAI/IAAI. Cambridge: MIT Press. S. 700-705.

Korf, R. E. (1999): *A Divide and Conquer Bidirectional Search: First results*. In: Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI '99). San Francisco: Morgan Kaufmann Publishers Inc. S. 1184-1191.

Korf, R. E. und Felner, A. (2002): *Disjoint Pattern Database Heuristics*. Artificial Intelligence 134(1-2). S. 9-22.

Korf, R. E. und Schultze, P. (2005): *Large-Scale Parallel Breadth-First Search.* In: AAAI/IAAI. Cambridge: MIT Press. S. 1380-1385.

Korf, R. E., Zhang, W., Thayer, I. und Hohwald, H. (2005): *Frontier search.* Journal of the ACM 52(5). S 715-748.

Kumar, V. und Rao, V. N. (1990): *Scalable parallel formulations of depth-first search.* In: Kumar, V., Gopalakrishnan, P. S. und Kanal, L. N. (Hrsg.): Parallel Algorithms for Machine Intelligence and Vision. New York: Springer. S. 1-41.

Pearl, J. (1984): *Heuristics. Intelligent Search Strategies for Computer Problem Solving.* Reading: Addison Wesley Publishing Company.

Powley, C., Ferguson, C. und Korf, R. E. (1990): *Parallel heuristic search: Two approaches.* In: Kumar, V., Gopalakrishnan, P. S. und Kanal, L. N. (Hrsg.): Parallel Algorithms for Machine Intelligence and Vision. New York: Springer. S. 42–65.

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G. und Kozyrakis, C. (2007): *Evaluating MapReduce for Multi-core and Multiprocessor Systems.* In: Proceedings of the 13th International Symposium on High-Performance Computer Architecture (HPCA '07). Phoenix: IEEE Computer Society. S. 13–24.

Reinefeld, A. und Schnecke, V. (1994): *AIDA* – Asynchronous Parallel IDA*.* In: Proceedings of the 10th Canadian Conference on Artificial Intelligence (AI'94). S. 295–302.

Reinefeld, A. und Schütt, T. (2009): *Out-of-Core Parallel Frontier Search with MapReduce.* In: Mewhort, D. J. K., Cann, N. M., Slater, G. W. und Naughton, T. J. (Hrsg.): High Performance Computing Systems and Applications. Lecture Notes in Computer Science 5976. New York: Springer. S. 323-336.

Reinefeld, A., Schütt, T. und Maier, R. (2010): *Very large pattern databases for heuristic search.* In: Hariri, S. und Keahey, K. (Hrsg.): Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC 2010). Chicago: ACM. S. 803-809.

Schwan, P. (2003): *Lustre. Building a file system for 1,000-node clusters.* In Proceedings of the 2003 Linux Symposium. Ottawa. S. 380–386.

Simon, H. A. und Newell, A. (1972): *Human Problem Solving.* Englewood Cliffs: Prentice-Hall.

Yang, F., Culberson, J. C., Holte, R., Zahavi, U. und Felner, A. (2008): *A general theory of additive state space abstractions* Journal of Artificial Intelligence Research 32. S 631-662.

Zahavi, U., Felner, A., Holte, R. C. und Schaeffer, J. (2008): *Duality in Permutation State Spaces and the Dual Search Algorithm.* Artificial Intelligence 172(4-5). S. 514-540.

Zhang, Y. und Hansen, E. (2006): *Parallel breath-first heursitic search on a shared-memory architecture.* Workshop on Heuristic Search, Memory-Based Heuristics and Their Applications. Menlo Park: AAAI Press.

Zhou, R. und Hansen, E. A. (2006): *Breadth-first heuristic search.* Artificial Intelligence 170(4–5). S. 385–408.

A. Anhang

A.1. Methodischer Anhang

Versuchsaufbau

In Reinefeld et al. (2010) haben wir die Leistung aller 127 möglichen dual-additiver PDBs des 8er-Puzzles analysiert. Die Untersuchung der Gruppen soll auf dieser Menge PDBs basieren. Für $n = 1...10$ wurden Gruppen der mit je n PDBs durch eine Zufallsziehung aus der Menge der dual-additiven PDBs gebildet. Pro Gruppenstärke wurden auf 10000 Gruppen bestimmt, wobei für die Analyse nur solche Berücksichtigung fanden, deren Speichergröße kleiner ist als die perfekte Heuristik (181440 Bytes). Zusätzlich dazu wurden alle einzelnen dual-additiven PDBs, sowie die perfekte Heuristik h^* evaluiert. Die genauer Zahl der Gruppen N ist je Gruppenstärke $|G|$ in Tabelle A.1 angegeben.

Um die Leistung der 60513 Gruppen festzustellen, wurden jeweils alle 181440 Positionen des 8er-Puzzles mit BFIDA* gelöst. Anstatt die einzelnen Probleme parallel zu lösen, wurden mehrere Probleme gleichzeitig mit je einem Prozessor gelöst. Erfasst wurden neben der exakten Gruppenzusammensetzung, die Summe der expandierten Knoten, die zur Lösung aller Problemstellungen erforderlich war, sowie der über alle Probleme gemittelte Leistungsverlust gegenüber der perfekten Heuristik und dessen Standardabweichung.

Zusätzlich dazu wurde der selbe Versuche unter Berücksichtigung Spiegelung an der Hauptdiagonalen durchgeführt.

Rangbildung

Die PDB mit der niedrigsten Summe expandierter Knoten erhält den Rang 1. Expandieren n PDBs dieselbe Anzahl an Knoten, erhalten all diese PDBs das arithmetische Mittel der ihnen zugeordneten Ränge.

$\|G\|$	1	2	3	4	5	6	7	8	9	10	$\sum$
N	128	9753	9533	9054	8265	7301	6197	4517	3233	2172	60153

Tabelle A.1.: Zahl der Gruppen N je Gruppengröße $|G|$.

Detailierte Daten zum effizienten Rand

Die Daten zu dem in Abbildung 4.5 dargestellten effizienten Rand sind in Tabelle A.2 aufgelistet. Die erste Spalte gibt die Speichergröße der Gruppe wider, in der zweiten Spalte steht die Summe der expandierten Knoten. Die nachfolgenden Spalten geben die Zahl der PDBs vom jeweiligen Typ in der Gruppe an.

Größe	Knoten	4+4	5+3	6+2	7+1	8+0
6048	32.492.604	1	0	0	0	0
12096	20.945.858	2	0	0	0	0
18144	18.178.381	3	0	0	0	0
21672	17.895.856	1	1	0	0	0
24192	16.737.382	4	0	0	0	0
27720	16.463.347	2	1	0	0	0
30240	15.798.802	5	0	0	0	0
33768	15.549.883	3	1	0	0	0
37296	15.375.880	1	2	0	0	0
39816	15.173.393	4	1	0	0	0
42336	15.070.843	7	0	0	0	0
43344	14.856.867	2	2	0	0	0
45864	14.353.564	5	1	0	0	0
51912	14.050.430	6	1	0	0	0
55440	14.028.260	4	2	0	0	0
57960	13.700.325	7	1	0	0	0
61488	13.483.872	5	2	0	0	0
71064	13.238.620	4	3	0	0	0
72648	12.596.575	2	0	1	0	0
78696	12.472.985	3	0	1	0	0
84744	12.279.337	4	0	1	0	0
88272	12.164.781	2	1	1	0	0
90792	11.931.040	5	0	1	0	0
94320	11.674.940	3	1	1	0	0
109944	11.616.190	3	2	1	0	0
112464	11.561.503	6	1	1	0	0
118512	11.232.275	7	1	1	0	0
127152	11.175.551	1	0	2	0	0
131049	10.330.283	0	0	0	1	0
137097	9.768.864	1	0	0	1	0
143145	9.529.550	2	0	0	1	0
149193	9.488.489	3	0	0	1	0
152721	9.447.410	1	1	0	1	0
158769	9.355.204	2	1	0	1	0
161289	9.339.589	5	0	0	1	0
167337	9.300.977	6	0	0	1	0
168345	9.287.029	1	2	0	1	0
174393	9.261.664	2	2	0	1	0
179433	9.178.715	8	0	0	1	0
181440	7.833.973	0	0	0	0	1

Tabelle A.2.: Daten des in Abbildung 4.5 dargestellten effizienten Rands.

A.2. Zehn 6+6+6+6 Partitionierung für das 24er-Puzzle

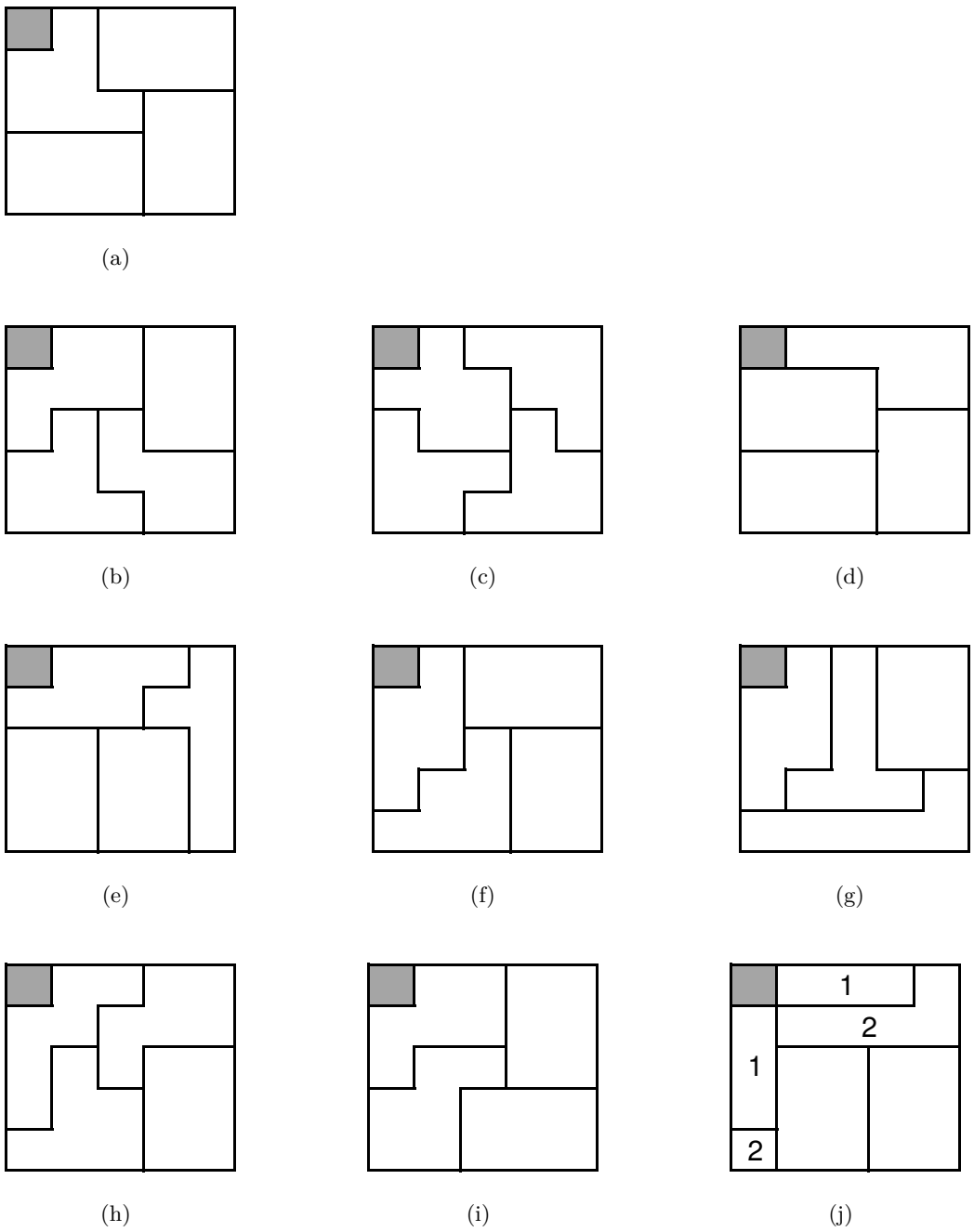


Abbildung A.1.: Verschiedene 6+6+6+6 Partitionierungen für das 24er-Puzzle. Die PDB in A.1(a) stammt aus Korf und Felner (2002). Die restlichen sind von Hand erstellt worden.

A.3. 50 Probleminstanzen des 24er-Puzzles

ID	Position	Tiefe
38	10 3 24 12 0 7 8 11 14 21 22 23 2 1 9 17 18 6 20 4 13 15 5 19 16	96
40	2 17 4 13 7 12 10 3 0 16 21 24 8 5 18 20 15 19 14 9 22 11 6 1 23	82
25	3 17 9 8 24 1 11 12 14 0 5 4 22 13 16 21 15 6 7 10 20 23 2 18 19	81
32	1 12 18 13 17 15 3 7 20 0 19 24 6 5 21 11 2 8 9 16 22 10 4 23 14	97
44	8 12 18 3 2 11 10 22 24 17 1 13 23 4 20 16 6 15 9 21 19 5 14 0 7	93
37	23 22 5 3 9 6 18 15 10 2 21 13 19 12 20 7 0 1 16 24 17 4 14 8 11	100
30	8 19 7 16 12 2 13 22 14 9 11 5 6 3 18 24 0 15 10 23 1 20 4 17 21	92
13	21 24 8 1 19 22 12 9 7 18 4 0 23 14 10 6 3 11 16 5 15 2 20 13 17	101
1	14 5 9 2 18 8 23 19 12 17 15 0 10 20 4 6 11 21 1 7 24 3 16 22 13	95
28	17 15 7 12 8 3 4 9 21 5 16 6 19 20 1 22 24 18 11 14 23 10 2 13 0	98
36	2 10 1 7 16 9 0 6 12 11 3 18 22 4 13 24 20 15 8 14 21 23 17 19 5	90
5	17 1 20 9 16 2 22 19 14 5 15 21 0 3 24 23 18 13 12 7 10 8 6 4 11	100
22	7 6 3 22 15 19 21 2 13 0 8 10 9 4 18 16 11 24 5 12 17 1 23 14 20	95
16	18 24 17 11 12 10 19 15 6 1 5 21 22 9 7 3 2 16 14 4 20 23 0 8 13	96
29	10 3 6 13 1 2 20 14 18 11 15 7 5 12 9 24 17 22 4 8 21 23 19 16 0	88
4	18 14 0 9 8 3 7 19 2 15 5 12 1 13 24 23 4 21 10 20 16 22 11 6 17	98
26	22 21 15 3 14 13 9 19 24 23 16 0 7 10 18 4 11 20 8 2 1 6 5 17 12	105
3	6 0 24 14 8 5 21 19 9 17 16 20 10 13 2 15 11 22 1 3 7 23 4 18 12	97
31	19 20 12 21 7 0 16 10 5 9 14 23 3 11 4 2 6 1 8 15 17 13 22 24 18	99
41	13 19 9 10 14 15 23 21 24 16 12 11 0 5 22 20 4 18 3 1 6 2 7 17 8	106
47	21 11 10 4 16 6 13 24 7 14 1 20 9 17 0 15 2 5 8 22 3 12 18 19 23	92
27	9 19 8 20 2 3 14 1 24 6 13 18 7 10 17 5 22 12 21 16 15 0 23 11 4	99
43	7 4 19 12 16 20 15 23 8 10 1 18 2 17 14 24 9 5 0 21 6 3 11 13 22	104
46	1 16 10 14 17 13 0 3 5 7 4 15 19 2 21 9 23 8 12 6 11 24 22 20 18	100
45	9 7 16 18 12 1 23 8 22 0 6 19 4 13 2 24 11 15 21 17 20 3 10 14 5	101
6	2 0 10 19 1 4 16 3 15 20 22 9 6 18 5 13 12 21 8 17 23 11 24 7 14	101
7	21 22 15 9 24 12 16 23 2 8 5 18 17 7 10 14 13 4 0 6 20 11 3 1 19	104
49	2 21 3 7 0 8 5 14 18 6 12 11 23 20 10 15 17 4 9 16 13 19 24 22 1	100
35	2 10 24 11 22 19 0 3 8 17 15 16 6 4 23 20 18 7 9 14 13 5 12 1 21	98
8	7 13 11 22 12 20 1 18 21 5 0 8 14 24 19 9 4 17 16 10 23 15 3 2 6	108
39	16 24 3 14 5 18 7 6 4 2 0 15 8 10 20 13 19 9 21 11 17 12 22 23 1	104
23	24 11 18 7 3 17 5 1 23 15 21 8 2 4 19 14 0 16 22 6 9 13 20 12 10	104
15	24 10 15 9 16 6 3 22 17 13 19 23 21 11 18 0 1 2 7 8 20 5 12 4 14	103
2	16 5 1 12 6 24 17 9 2 22 4 10 13 18 19 20 0 23 7 21 15 11 8 3 14	96
19	16 13 6 23 9 8 3 5 24 15 22 12 21 17 1 19 10 7 11 4 18 2 14 20 0	106
42	16 6 20 18 23 19 7 11 13 17 12 9 1 24 3 22 2 21 10 4 8 15 14 5 0	108
20	4 5 1 23 21 13 2 10 18 17 15 7 0 9 3 14 11 12 19 8 6 20 24 22 16	92
24	14 24 18 12 22 15 5 1 23 11 6 19 10 13 7 0 3 9 4 17 2 21 16 20 8	107
17	23 16 13 24 5 18 22 11 17 0 6 9 20 7 3 2 10 14 12 21 1 19 15 8 4	109
34	5 18 3 21 22 17 13 24 0 7 15 14 11 2 9 10 1 8 6 16 19 4 20 23 12	102
48	2 22 21 0 23 8 14 20 12 7 16 11 3 5 1 15 4 9 24 10 13 6 19 17 18	107
12	12 23 9 18 24 22 4 0 16 13 20 3 15 6 17 8 7 11 19 1 10 2 14 5 21	109
21	24 8 14 5 16 4 13 6 22 19 1 10 9 12 3 0 18 21 20 23 15 17 11 7 2	103
18	0 12 24 10 13 5 2 4 19 21 23 18 8 17 9 22 16 11 6 15 7 3 14 1 20	110
33	11 22 6 21 8 13 20 23 0 2 15 7 12 18 16 3 1 17 5 4 9 14 24 10 19	106
14	24 1 17 10 15 14 3 13 8 0 22 16 20 7 21 4 12 9 2 11 5 23 6 18 19	111
10	23 14 0 24 17 9 20 21 2 18 10 13 22 1 3 11 4 16 6 5 7 12 8 15 19	114
11	15 11 8 18 14 3 19 16 20 5 24 2 17 4 22 10 1 13 9 21 23 7 6 12 0	106
9	3 2 17 0 14 18 22 19 15 20 9 7 10 21 16 6 24 23 8 5 1 4 11 12 13	113
50	23 1 12 6 16 2 20 10 21 18 14 13 17 19 22 0 15 24 3 7 4 8 5 9 11	113

Tabelle A.3.: Zufallsinstanzen des 24er-Puzzles aus Korf und Felner (2002) sortiert nach der Knotenexpansion von IDA*.