# Effiziente speicherplatzbeschränkte Graph-Such-Algorithmen*

Jürgen Eckerle und Sven Schuierer†

**Zusammenfassung**

Heuristische Suchalgorithmen sind Verfahren, die in einem Graphen unbekannter, in der Regel exponentieller Größe nach optimalen Wegen zu einem oder mehreren vorgebenen Zielknoten suchen. Sie verwenden dabei problemspezifisches Wissen, das durch eine Heuristik zur Verfügung gestellt wird.

In diesem Bericht stellen wir heuristische Suchalgorithmen vor, die einen vorgebenen, beschränkten Speicherplatz der Größe $m$ verwenden. Als Maß für die Güte eines Verfahrens $A$ betrachten wir die Anzahl der von $A$ expandierten Knoten im Verhältnis zu der Anzahl $n$ von Knoten, die das Verfahren A* expandiert, um das gleiche Problem zu lösen. Die bisher in der Literatur vorgestellten Verfahren wie IDA*, IDA*_CR, MA*, SMA* etc. expandieren bis zu $\Omega(n^2)$ Knoten und weisen einen hohen Overhead auf, der diese Verfahren für praktische Anwendungen ungeeignet macht. Für unsere Verfahren kann gezeigt werden, daß die Anzahl der expandierten Knoten durch $O(dn^2/m)$ bzw. $O(n^2/m)$ beschränkt ist, wobei $d$ die Länge eines Lösungsweges und $m$ die Größe des benötigten Speichers ist. Bei Verwendung eines entsprechend großen Speichers kann dies zu einer deutlichen Verringerung der Anzahl der expandierten Knoten führen. Zudem zeichnen sich diese Algorithmen durch eine wesentlich geringere Anzahl von Update-Operationen der benutzten Datenstrukturen aus, was in praktischen Anwendungen die Laufzeit erheblich reduzieren kann.

*Diese Arbeit wurde durch die Deutsche Forschungsgemeinschaft unterstützt unter der Nr. Ot 64/8-1.

†Institut für Informatik, Universität Freiburg, Rheinstr. 10, D-79104 Freiburg

# 1 Einführung

Ein zentrales Gebiet der KI stellt die heuristische Suche dar. Eine typische Anwendung für heuristische Suchalgorithmen ist z.B. das Lösen von Puzzles. Hier besteht die Aufgabe darin, von einer gegebenen Startstellung durch Ausführung erlaubter Züge (Operationen) in eine von möglicherweise mehreren Zielstellungen zu kommen. Um eine Zielstellung zu erreichen, kann man dabei nach dem *trial-and-error* Prinzip vorgehend jedes Mal einen der möglichen Züge ausführen. Diese Methode ist jedoch i.a. nicht praktikabel, da selbst bei so einfachen Puzzles wie Sam Loyds Fünfzehnerpuzzle, der zu durchsuchende Lösungsraum exponentiell mit der Lösungstiefe wächst. Hier setzen heuristische Suchverfahren an, die dank der Anwendung problemspezifischen Wissens das unnötige Durchsuchen großer Teile des Lösungsraumes vermeiden.

Formal geht man dabei von einer Menge von Problemzuständen $Z$ und und einer Menge $O$ von Operatoren aus, deren Anwendung einen Zustand in einen anderen überführt. Ein Problem zu lösen heißt nun, eine Folge von Operatoren $o_1, o_2, \ldots, o_d$ zu finden, die nacheinander auf den Startzustand $s \in Z$ angewandt einen Zielzustand $g$ liefern. Die Anzahl $d$ der ausgeführten Operationen ist die *Lösungstiefe*. Oft wird eine möglichst „günstige“ Lösung gesucht, d.h. belegt man die Ausführung eines Operators $o_i$ mit den Kosten $c(o_i)$, so soll die Summe $\sum_{i=1}^{d} c(o_i)$ minimiert werden. Eine Lösung mit minimalen Kosten heißt *optimale Lösung*.

Häufig verwendete Kriterien, um heuristische Suchalgorithmen zu bewerten, sind die Zeitkomplexität, die Speicherkomplexität und die Güte der gefundenen Lösung. Wir messen die Zeitkomplexität, wie auch sonst in der Literatur üblich, durch die Anzahl der expandierten Knoten des den Lösungsraum repräsentierenden, implizit gegebenen Graphen. Andere Faktoren wie z.B. die Zugriffszeiten auf die verwendeten Datenstrukturen werden nicht berücksichtigt. Das wohl bekannteste heuristische Suchverfahren ist das A*-Verfahren [6]. A* ist ein *zulässiges* Verfahren, d.h. es garantiert das Auffinden einer optimalen Lösung. A* besitzt den Nachteil, daß der Speicherplatzbedarf exponentiell mit der Lösungstiefe wächst. Dieser Nachteil wird durch das Verfahren IDA* vermieden, ebenfalls ein zulässiges Verfahren, mit dem es erstmals gelang, zufällig gewählte Probleme des Fünfzehnerpuzzles zu lösen [3]. IDA$^*$ speichert nur den aktuellen Suchpfad, hat also einen zur Lösungstiefe proportionalen Speicherplatzbedarf. Diese Speicherplatzersparnis geht jedoch auf Kosten der Laufzeitkomplexität, denn, falls A$^*$ $n$ Knoten expandiert, so expandiert IDA$^*$ im schlechtesten Fall bis zu $\Omega(n^2)$ Knoten.

Daher richtet sich die Suche auf (zulässige) Algorithmen, die die Vorteile von A$^*$ (geringe Anzahl expandierter Knoten) und die von IDA$^*$ (geringer Speicherplatzbedarf) vereinen. Dies ist ein in der Literatur bereits ausführlich untersuchtes Problem [1, 4, 5, 9, 10, 11, 12, 14], wobei alle genannten Arbeiten Algorithmen vorstellen, die mit einem beschränktem Speicherplatz arbeiten. Die Algorithmen weisen teilweise eine große Ähnlichkeit untereinander auf und lassen sich in mehrere Gruppen unterteilen.

Die erste Gruppe besteht aus den Algorithmen RBFS [4] und IE [10]. Sie speichern nur den gerade aktuellen Suchpfad. Der Speicherplatz wird dadurch nur zu einem geringen Bruchteil ausgenutzt. Die Komplexität dieser Verfahren liegt wie im Fall von IDA$^*$ bei $\Omega(n^2)$, wenn $n$ die Anzahl der von A$^*$ expandierten Knoten ist.

Die zweite Gruppe besteht aus MA$^*$ [1], SMA$^*$ [10] und ITS [5]. Wir betrachten hier stellvertretend für die anderen den Algorithmus MA$^*$. MA$^*$ speichert neben dem aktuellen Suchpfad bevorzugt die Knoten in der unmittelbaren Nachbarschaft der zuletzt expandierten Knoten. Dies führt dazu, daß MA$^*$ nur dann befriedigend arbeitet, wenn mit hoher Wahrscheinlichkeit der nächste zu expandierende Knoten in unmittelbarer Nachbarschaft des zuletzt expandierten liegt. Ist diese Voraussetzung nicht gegeben, hat MA$^*$ dieselbe Komplexität wie IDA$^*$. Ein weiterer Nachteil besteht in dem hohen Overhead, der das Verfahren sehr ineffizient und für die Praxis unbrauchbar macht. R. E. Korf erwähnt, daß das Verfahren angewandt auf das Fünfzehnerpuzzle 20 mal langsamer ablief als IDA$^*$. Auch eine von den Autoren selbst vorgenommene Implementierung zeigte beim Fünfzehnerpuzzle keine wesentliche Reduzierung der Anzahl der Knotenexpansionen gegenüber IDA$^*$, insbesondere verringerte sich die Komplexität von MA$^*$ nicht umgekehrt proportional zur verwendeten Speicherplatzgröße $m$. SMA$^*$ und ITS stellen Verbesserungen gegenüber MA$^*$ dar, mit dem Ziel den Overhead zu reduzieren. Aber auch diese haben ähnliche prinzipielle Nachteile wie der Algorithmus MA$^*$.

Die dritte Gruppe wird gebildet von den Algorithmen IDA*_CR [11], DFS* [9] und MIDA* [14], die sich ebenfalls sehr ähnlich sind. IDA*_CR reduziert bei einigen Anwendungen, wie z. B. dem euklidischen Traveling-Salesman-Problem, die Anzahl der expandierten Knoten erheblich, nutzt aber zur Verfügung stehende Speicherplatzressourcen nicht zu einer Verbesserung der Komplexität aus. Ferner kann es sein, daß dieser Algorithmus gerade im letzten Schritt die meisten Knoten expandiert, während A* nur einen einzigen Knoten expandiert. Auch hier werden bis zu $\Omega(n^2)$ Knoten expandiert.

Wir wollen hier noch MREC [12] erwähnen, ein Verfahren das zunächst A* durchführt, bis der zur Verfügung stehende Speicherplatz belegt ist. Um weitere Knoten zu expandieren, wird von den Blättern des gespeicherten Graphen der IDA*-Algorithmus gestartet. Mit Ausnahme des aktuellen Suchpfades wird jedoch der einmal gespeicherte Teilgraph nicht mehr verändert. Dieses statische Vorgehen bedingt wiederum eine Worst-case Komplexität von $\Omega(n^2)$.

In diesem Artikel stellen wir einen neuen Ansatz vor, wie der zur Verfügung stehende Speicherplatz effizient zur Verringerung der Anzahl der Knotenexpansionen verwendet werden kann. Dabei wird die Speicherplatzausnutzung dynamisch der Struktur des Suchbaumes angepaßt, indem die gespeicherten Knoten gleichmäßig über den gesamten Suchbaum verteilt werden. Die dafür entwickelten Methoden lassen sich auch mühelos mit allen hier erwähnten Algorithmen kombinieren, was zu einer weiteren Reduktion der Anzahl der Knotenexpansionen führen kann. Bei den in dieser Arbeit vorgestellten Algorithmen vermindert sich die Anzahl der Knotenexpansionen umgekehrt proportional zum vorhandenen Speicherplatz $m$. Es gilt genauer: Bei einem vorgegeben Speicherplatz der Größe $m$ werden höchstens $O(dn^2/m)$ bzw. $O(n^2/m)$ Knoten expandiert, wobei $d$ die Länge eines optimalen Weges bezeichnet. Dies stellt eine deutliche Verbesserung der bisher bekannten Resultate dar. Ferner lassen sich unsere Verfahren sehr effizient implementieren.

Die Arbeit gliedert sich folgendermaßen. Im 2. Abschnitt definieren wir grundlegende Begriffe wie Graph, Suchalgorithmus, Heuristik usw. und stellen die klassischen Verfahren A* und IDA* vor. Im 3. Abschnitt legen wir die bisherigen Ansätze kurz dar, die einen beschränkten Speicherplatz verwenden und diskutieren ihre Schwächen. Unsere Algorithmen stellen wir im 4. Abschnitt vor. Zunächst zeigen wir eine einfache Variante des IDA*-Verfahrens und gehen danach auf ein komplizierteres Verfahren zur Baumaufteilung ein.

# 2 Grundlagen

## 2.1 Graph, Suchbaum

Wir betrachten Einpersonenspiele in dem allgemeinen Rahmen von *state-space* Problemen. Ein *state-space* Problem ist gegeben durch eine Menge von Zuständen $Z$, eine Menge von Operatoren $O \subseteq Z \times Z$, einem ausgezeichneten Element $s \in Z$, das *Startzustand* heißt, und einer Menge von Zielzuständen $T$. Ein Operator $o = (z_1, z_2) \in O$ überführt einen Zustand $z_1$ in einen Zustand $z_2$. Eine Gewichtsfunktion $c : O \longrightarrow \mathbb{R}^+$ ordnet jedem Operator $o \in O$ die Kosten seiner Anwendung zu. Ein *state-space* Problem läßt sich mit dessen Repräsentation als Graph identifizieren. Knoten entsprechen Zuständen, Operatoren gewichteten Pfeilen. Genauer gilt: Der einem *state-space* Problem zugeordnete Graph wird durch das Tupel

$$G = (V, E, s, T, c),$$

beschrieben, wobei $V$ die Menge der Knoten, $E$ die Menge der gerichteten Kanten (Pfeile), $s \in V$ der Startknoten, $T$ die Menge der Zielknoten und $c : E_G \longrightarrow \mathbb{R}^+$ die Kostenfunktion ist. Der *Nachfolgeroperator* $\Gamma$ ordnet jedem $x \in V$ die Menge derjenigen $y \in V$ zu, zu denen eine gerichtete Kante von $x$ nach $y$ führt, d. h. für die $(x, y) \in E$ ist; $y$ heißt *Sohn* (*Nachfolger*) von $x$, $x$ *Vater* (Vorgänger) von $y$. Ein *Weg* in $G$ ist eine Folge $w : s = x_0, x_1, \ldots, x_k$, mit $x_{i+1} \in \Gamma(x_i)$, für $0 \leq i < k$. Die *Kosten* (das *Gewicht*) dieses Weges ist durch die Summe $c(w) = \sum_{i=1}^{k-1} c(x_i, x_{i+1})$ gegeben. Gibt es keinen Weg $x_0, x_1, \ldots, x_k$ mit $x_0 = x_k$ so nennen wir $G$ *azyklisch*. Ein wichtige Klasse von Graphen ist die der Bäume. Ein azyklischer Graph heißt *Baum*, wenn es ein ausgezeichnetes Element $r$ gibt mit der Eigenschaft, daß es zu jedem $x \in V$ genau einen Weg von $r$ nach $x$ gibt. Der Knoten $r$ heißt *Wurzel* von $G$. Die *Tiefe* eines Knotens $x$ ist die Länge eines Weges von $r$ nach $x$. Knoten ohne Nachfolger heißen *Blätter* oder *Horizontknoten*.

Ein Problem zu *lösen*, heißt nun einen Weg von $s$ nach $g$ mit $g \in T$ zu finden. Eine *optimale Lösung* ist ein Weg mit minimalen Kosten. Belegt z.B. die Kostenfunktion jeden Pfeil mit dem Wert 1, so ist eine optimale Lösung ein Lösungsweg mit minimaler Länge. Der aus einem *state-space* Problem resultierende Graph ist ein OR-Graph, d. h. daß die Nachfolger eines Problemzustandes $x$ alternativ zu lösende Teilprobleme darstellen. Heuristische Suchverfahren werden bei dieser Sichtweise als Graphenalgorithmen aufgefaßt.

Wir unterscheiden zwei Repräsentationen eines Graphen. Unter der *expliziten* Repräsentation eines Graphen verstehen wir die Repräsentation des Graphen durch Angabe aller Knoten und Kanten (einschließlich der Kosten). Die *implizite* Repräsentation eines Graphen ist durch Angabe eines Startzustandes $s$, einer Zielmenge $T$ und eines Nachfolgeroperators $\Gamma$ gegeben. Einen Knoten $x$ zu *expandieren*, heißt für den Knoten $x$ die Menge $\Gamma(x)$ zu erzeugen. Eine Lösung zu finden, heißt einen Teilgraphen zu expandieren, der einen Lösungsweg enthält.

Aus einer maschinennäheren Sichtweise kann jeder Suchalgorithmus als eine Folge von Anwendungen des Nachfolgeoperators $\Gamma$ und von Operationen auf Datenstrukturen verstanden werden, die eine Teilmenge der bereits expandierten Knoten verwalten. Diese Operationen sind typischerweise Minimumbestimmung, Einfügen, Entfernen und Suchen. Sie steuern die Entscheidung, welcher Knoten als nächster expandiert werden soll.

Am Beispiel des Fünfzehnerpuzzles wird das Problem deutlich, daß verschiedene Zugfolgen zu derselben Spielstellung führen können. Wir tragen in der folgenden Definition des *traversal tree* dieser Tatsache Rechnung, indem wir Spielstellungen entsprechend ihrer Entstehungsgeschichte unterscheiden. Sei $G$ ein Graph. Die Knoten des *traversal tree* $T_G$ von $G$ sind durch die Menge der Wege in $G$ mit Startknoten $s$ gegeben. Der Knoten $(s = x_0, x_1, \cdots, x_k)$ ist Vaterknoten von $(s = x_0, x_1, \cdots, x_k, x_{k+1})$, wenn $x_{k+1}$ ein Vorgänger von $x_k$ in $G$ ist (d. h. $x_{k+1} \in \Gamma(x_k)$). Offensichtlich ist $T_G$ ein Baum. Der Endknoten eines Weges ist ein Knoten aus $G$ und wird durch den *end* Operator geliefert, d.h. $end(x_0, x_1, \cdots, x_k) = x_k$. Jeder Knoten $(s = x_0, x_1, \cdots, x_k)$ aus $T_G$ liefert offenbar auf eindeutige Weise eine Expansionsfolge, die die Erzeugung des Knoten $x_k$ aus $G$ wiedergibt.

Jeder Suchalgorithmus läßt sich als Folge von endlichen Teilbäumen von $T_G$ auffassen, d.h. ein Algorithmus $A$ erzeugt und speichert im Verlauf seiner Abarbeitung Suchbäume $T_i$, die Teilbäume von $T_G$ sind.

Wird eine bereits erzeugte Spielstellung $x \in G$ im Schritt $i$ von $A$ zum zweiten Mal (durch eine andere Zugfolge) erreicht, so ist es sinnvoll, nur den Weg mit den kleineren Kosten in $T_i$ zu speichern. In $T_i$ besteht also eine eindeutige Beziehung zwischen den Knoten von $T_i$ und den diese repräsentierenden Spielstellungen. Der Einfachheit halber wollen wir daher die Knoten von $T_i$ mit ihren Spielstellungen identifizieren und dies auch für $T_G$ übernehmen, sofern dies nicht zu Mißverständnissen führen kann.

Mit $path(x)$ bezeichnen wir den eindeutig bestimmten Weg, der zur Erzeugung von $x \in G$ (bzw. $x \in T_i$) führt, und mit $g(x)$ die Kosten des Weges $path(x) = (s = x_0, x_1, \cdots, x_k)$, definiert durch $g(x) = \sum_{i=0}^{k} c(x_i, x_{i+1})$. Weiterhin sei $p(x)$ der Vaterknoten von $x$ in $T_i$, und $S(x)$ die Menge der Sohnknoten von $x$ in $T_i$.

Ist die Häufigkeit, mit der die Menge der Knoten aus $G$ als Endpunkte von Wegen in $T_G$ auftritt, durch $c \in \mathbb{R}^+$ beschränkt, d.h., gilt $|V_{T_G}| \leq c|V_G|$, so nennen wir den Graphen $G$ *c-beschränkt*.

## 2.2 Heuristiken

Heuristiken sind Funktionen, die die Suche nach einem Zielzustand erleichtern sollen, indem sie einem Suchalgorithmus Wissen über das Problem zur Verfügung stellen. Formal definieren wir eine *Heuristik* als eine Funktion $h : V \rightarrow \mathbb{R}^+$, mit $h(g) = 0$ für $g \in T$. Wir interpretieren $h(x)$ als Abschätzung der Kosten eines Weges von einem die Stellung $x$ repräsentierenden Knoten in $T_G$ zu einem Zielknoten. Im folgenden seien $h^*(x)$ die minimalen Kosten eines solchen Pfades von $x$ zu einem Zielzustand. Ist $h(x) \leq h^*(x)$, für alle $x \in G$, so heißt die Heuristik $h$ *optimistisch* (engl.: admissible). Gilt für alle $x, y \in G$, mit $(x, y) \in E$ die Dreiecksungleichung $h(x) \leq h(y) + c(x, y)$, so heißt $h$ *konsistent* (oder *monoton*) (engl.: consistent). Wie man leicht sieht, ist jede konsistente Heuristik auch optimistisch.

Die Eigenschaften einer Heuristik beeinflussen häufig die Eigenschaften eines Suchalgorithmus, so gilt z.B., daß das Verfahren A* (siehe weiter unten) genau dann immer mit einer optimalen Lösung terminiert—vorausgesetzt, daß überhaupt eine Lösung existiert—wenn die verwendete Heuristik optimistisch ist [8].

Im folgenden betrachten wir Graphen immer in Verbindung mit einer Heuristik. Daher führen wir den Begriff der Probleminstanz ein. Eine *Probleminstanz* $I$ ist ein Paar $I = (G, h)$, bestehend aus einem Graphen $G$ und einer Heuristik $h$ auf $G$. Für uns sind sind die folgenden Klassen von Probleminstanzen wichtig:

$$\begin{aligned} I_{AD} &:= \{I = (G, h) \mid G \text{ Graph, } h \text{ Heuristik, } h \text{ ist optimistisch} \\ &\qquad \text{(engl.: ADmissible) auf } G\} \\ I_{CON} &:= \{I = (G, h) \mid G \text{ Graph, } h \text{ Heuristik, } h \text{ ist konsistent} \\ &\qquad \text{(engl.: CONsistent) auf } G\} \end{aligned}$$

## 2.3 Heuristische Suchalgorithmen

Im folgenden betrachten wir Algorithmen, die mit Hilfe von Heuristiken den Problemgraphen durchsuchen. Eine der wichtigsten Forderungen an derartige Algorithmen ist, daß sie eine (fast) optimale Lösung liefern. Wir nennen einen Algorithmus *zulässig*, wenn dieser immer mit einer optimalen Lösung terminiert, vorausgesetzt, daß überhaupt eine Lösung existiert.

Heuristische Suchalgorithmen zielen darauf ab, den Suchbaum dort zu expandieren, wo das Finden einer Lösung am vielversprechendsten ist. Dazu ordnet man jedem Knoten $x$ aus $T_G$ einen Funktionswert $f(x)$ zu. Man nimmt nun an, daß von zwei Knoten derjenige mit kleinerem $f$-Wert eher in einem optimalen Pfad enthalten ist. Setzt man z.B. $f(x) = g(x)$ und expandiert immer den Knoten mit kleinstem $f$-Wert, so erhält man Dijkstras kürzesten Pfad Algorithmus. Hierbei wird das zusätzliche, von der Heuristik zur Verfügung gestellte Wissen nicht berücksichtigt.

Die von einem Algorithmus erzeugten Knoten werden in die Mengen *OPEN* und *CLOSED* unterteilt. Die Menge *OPEN* enthält alle mindestens einmal erzeugten, aber noch nicht expandierten Knoten (Horizontknoten) des aktuellen Suchbaumes, und die Menge *CLOSED* enthält alle bereits expandierten (inneren) Knoten.

*Best-first geordnete* Algorithmen sind Algorithmen, die immer den Knoten aus $OPEN$ mit kleinstem $f$-Wert expandieren [4]. In der Literatur wurden bisher im wesentlichen *best-first* Algorithmen untersucht, die ebenso wie best-first geordnete Algorithmen immer den Knoten aus $OPEN$ mit kleinstem $f$-Wert expandieren und sich zudem noch dadurch auszeichnen, daß beide Mengen $OPEN$ und $CLOSED$ explizit im Speicher vorliegen. Das A*-Verfahren ist ein spezieller best-first Algorithmus, den man erhält, wenn man für, $x \in V_G$,

$$f(x) = g(x) + h(x) \qquad (*)$$

setzt, wobei $g(x)$ und $h(x)$ wie oben definiert sind.

Im folgenden sei $f$ immer durch $(*)$ gegeben. Es läßt sich zeigen, daß ein best-first geordneter Algorithmus bei Verwendung einer optimistischen Heuristik immer zulässig ist. Wie wir im folgenden zeigen werden, ist es möglich, diese Bedingung weiter abzuschwächen und dennoch die Zulässigkeit von Algorithmen zu bewahren. Hierzu definieren wir $f^*(x) = \max\{f(y) \mid y \in path(x)\}$. Die Funktion $f^*$ liefert wie $f$ eine totale Ordnung auf der Menge der Knoten von $T_G$ (und damit natürlich auch auf der Menge der Knoten von $OPEN$ und $CLOSED$). Ein Algorithmus $A$ heißt *$f^*$-geordnet* oder kurz *geordnet*, falls zu jedem Zeitpunkt ein Knoten expandiert wird, dessen $f^*$-Wert kleiner oder gleich dem $f^*$-Wert aller Elemente aus $OPEN$ ist. Zunächst kann man leicht sehen, daß ein Algorithmus auf einer Probleminstanz aus $I_{CON}$ genau dann best-first geordnet ist, wenn er geordnet ist, da immer $f^*(x) = f(x)$ ist. Treten nicht-monotone Stellen auf so kann sich die Expansionsreihenfolge für die $f^*$- und die best-first Ordnung aber unterscheiden. Die best-first Ordnung ist im folgenden Sinne die stärkere Ordnung.

**Lemma 2.1** *Sei $A$ ein best-first geordneter Algorithmus auf Probleminstanz $(G, h)$ und $x_i$ der im Schritt $i$ expandierte Knoten aus der Menge der Horizontknoten $OPEN_i$, so gilt für alle $x \in OPEN_i$: $f^*(x) \geq f^*(x_i)$.*

**Beweis:** Sei $x_k \in path(x_i)$ so gewählt, daß $f(x_k) = f^*(x_i)$. Für einen Knoten $x$ aus $OPEN_i$ gilt: Es gibt einen Vorgänger $x'$ von $x$, der in $OPEN_k$ enthalten war. Für $x'$ ist offenbar $f(x') \geq f(x_k)$, da $x'$ in $OPEN_k$ enthalten ist und im Schritt $k$ $x_k$ expandiert wurde. Unter Berücksichtigung von $f^*(x) \geq f(x') \geq f(x_k) = f^*(x_i)$ folgt die Behauptung. □

Aus dem obigen Lemma folgt sofort, daß best-first geordnete Algorithmen auch $f^*$-geordnet sind. Die Umkehrung ist jedoch, wie leicht ersichtlich ist, falsch. Es ist klar, daß von zwei Knoten, die dieselbe Spielstellung repräsentieren, nur der günstiger erzeugte weiter expandiert werden muß. Die obigen Definitionen seien in diesem Sinne verallgemeinert.

Wie bereits erwähnt, ist die Definition eines geordneten Algorithmus dadurch motiviert, daß sie eine hinreichende Bedingung für die Zulässigkeit eines Suchverfahrens liefert. Denn:

**Satz 2.2** *Jeder geordnete Algorithmus ist auf einer Probleminstanz von $I_{AD}$ zulässig.*

**Beweis:** Es ist klar, daß ein geordneter Algorithmus $A$ terminiert, falls ein Lösungsweg existiert. Bleibt zu zeigen, daß $A$ in diesem Fall mit einer optimalen Lösung terminiert.

Wir führen einen Widerspruchsbeweis und nehmen an, daß es einen Graphen $G$ und eine optimistische Heuristik $h$ gibt, so daß $A$ angesetzt auf $G$ einen Zielknoten $x_A$ liefert, der nicht optimal ist, d. h. es existiert ein Zielknoten $x^*$ mit $g(x^*) < g(x_A)$.

Für jeden Knoten $x$ auf dem optimalen Lösungsweg $path(x^*)$ gilt

$$f(x) = g(x) + h(x) \leq g(x) + h^*(x) = g(x^*) = f(x^*) = f^*(x^*) < f^*(x_A),$$

da $h$ optimistisch ist.

Da zum Zeitpunkt der Terminierung von $A$ der Knoten $x^*$ noch nicht expandiert worden ist, existiert ein Knoten $x$ des Pfades $path(x^*)$, der in *OPEN* enthalten ist. Die obere Ungleichung gilt für alle Knoten auf dem Weg von $s$ nach $x^*$; insbesondere ist daher $f^*(x) < f^*(x_A)$ im Widerspruch zu der Annahme, daß $A$ geordnet ist und $x_A$ vor $x$ expandiert wird. □

Aus diesem Satz folgt z.B. sofort die Zulässigkeit des Verfahrens IDA* [6]. Die Umkehrung dieser Aussage gilt aber nicht: Ein Beispiel für einen zulässigen, aber nicht geordneten Algorithmus ist IDA*_CR [11].

Wir nennen einen Knoten $x$ des Transversal-Baums *$C$-beschränkt* bzw. *streng $C$-beschränkt*, falls $f^*(x) \leq C$ bzw. $f^*(x) < C$ ist. Sind $C^*$ die Kosten eines optimalen Lösungsweges, so folgt, daß alle $C^*$-streng beschränkten Knoten von A* sicher expandiert werden. Ob ein Knoten $x$ mit $f^*(x) = C^*$ expandiert wird hängt von der jeweiligen *tie-breaking*

Regel einer speziellen Implementation ab. Eine solche Regel gibt an, wie Unentschieden (engl. *ties*) gelöst werden, d. h. welcher von zwei Knoten mit demgleichen $f^*$-Wert zuerst expandiert wird.

J. Pearl hat gezeigt, daß der A*-Algorithmus optimal auf der Klasse $I_{CON}$ in dem Sinn ist, daß es zu jedem zulässigen Algorithmus $B$ und zu jeder Probleminstanz $I \in I_{CON}$ immer eine tie-breaking Regel gibt, so daß die Menge der von A* expandierten Knoten eine Teilmenge der Menge der von $B$ expandierten Knoten ist [8]. Für die Klasse $I_{AD}$ gibt es jedoch keinen in diesem Sinn optimalen Algorithmus. Auch über die Anzahl der expandierten Knoten kann keine Aussage getroffen werden. Dies zeigt, daß ein Vergleich der Komplexität zweier Algorithmen wesentlich von der Klasse der betrachteten Probleminstanzen abhängt. Daher beschränken wir uns, wie in der Literatur üblich, in dieser Arbeit auf die Klasse $I_{CON}$. Wir schreiben $N_A(I)$ für die Anzahl der von Algorithmus $A$ expandierten Knoten angesetzt auf $I$, und schreiben $N_{A1} = O(N_{A2})$, falls für alle $I \in I_{CON}$ $N_{A1}(I) = O(N_{A2}(I))$. Um einen fairen Vergleich der untersuchten Algorithmen mit dem A*-Verfahren hinsichtlich der Anzahl der Knotenexpansionen zu ermöglichen, müssen wir die möglichen tie-breaking-Regeln einschränken. So setzen wir hier generell voraus, das ein Unentschieden zugunsten desjenigen Knotens gelöst wird, der nach der Durchlaufordnung des Suchbaumes entsprechend der Hauptreihenfolge zuerst kommt. Hieraus läßt sich sofort folgern: Ist der State-space-Graph ein Baum, so erzeugt jeder $f^*$-geordnete Suchalgorithmus mit dieser tie-breaking-Regel denselben Teilbaum.

# 3 Klassische Verfahren

In diesem Abschnitt geben wir eine ausführliche algorithmische Beschreibung der beiden bekanntesten heuristischen Suchverfahren A* und IDA*.

## 3.1 Nomenklatur

| | |
|---|---|
| $G$ | (gerichteter, gewichteter) Graph |
| $\|G\|$ | die Anzahl der Knoten des Graphen $G$ |
| $s$ | Startknoten |
| $\Gamma(x)$ | die Menge der Nachfolgerknoten von $x$ |
| $S(x)$ | die Menge der Nachfolgerknoten von $x$, soweit sie nicht Sohnknoten von $x$ im bereits gespeicherten Teilbaum des erzeugten Suchbaumes enthalten sind |
| $T_G$ | der zu durchlaufende Suchbaum eines Graphen $G$ |
| $T^{(i)}$ | der im Schritt $i$ bereits erzeugte Teilgraph von $T_G$ |
| $U^{(i)}$ | der im Schritt $i$ vorliegende explizite Teilgraph von $T^{(i)}$ |
| $T_x^{(i)}$ | der Teilbaum in $T^{(i)}$ mit Wurzel $x$ |
| $d$ | die Lösungstiefe |
| $m$ | die Größe des zur Verfügung stehenden Speicherplatzes |

## 3.2 Das Verfahren A*

Wie weiter oben bereits erwähnt, ist der A*-Algorithmus ein best-first Suchalgorithmus, der alle erzeugten Knoten in den Listen *OPEN* und *CLOSED* speichert. Charakteristisch für A* ist die Tatsache, daß Duplikate beim Speichern der Knoten dadurch vermieden werden, daß nach jeder Expansion geprüft wird, ob ein neu erzeugter Knoten, genauer die durch diesen Knoten repräsentierte Spielstellung, bereits in *OPEN* oder in *CLOSED* enthalten ist. Falls ja, bleibt nur der Knoten gespeichert, der bisher am günstigsten erzeugt worden ist. Die Knoten des Suchbaumes werden dabei durch die folgende Pascal-ähnliche Typ-Deklaration beschrieben.

```
type
    Pointer = ↑Knoten;
    Knoten = record
                 pos: (Spiel-)Konfiguration; {die von diesem
                          Knoten repräsentierte Spielstellung}
                 cost: integer; {der f-Wert dieses Knotens}
                 pred: Pointer; {Zeiger auf den Vorgängerknoten}
```

**end;**

Die Operationen, die die Listen *OPEN* und *CLOSED* unterstützen müssen, führen wir im folgenden als Prozeduren oder Funktionen an.

Für *OPEN*:

**procedure** $initiate()$;
{initialisiert *OPEN*}
**function** $is\_empty()$: boolean;
{liefert, falls *OPEN* leer ist, den Wert *true* zurück, andernfalls *false*}
**procedure** $insert(x.cost, x.pos)$;
{fügt das Element $x$ entsprechend dem Wert $x.cost$ bzw. unter dem Suchschlüssel $x.pos$ in der als Vorwärtswarteschlange bzw. Suchstruktur aufgefaßten Datenstruktur für *OPEN* ein}
**function** $access\_min()$ : Knoten;
{liefert das minimale Element der Vorwärtswarteschlange}
**procedure** $delete\_min()$;
{entfernt das minimale Element der Vorwärtswarteschlange}
**procedure** $decrease\_key(p, a)$;
{vermindert den Wert $\uparrow p.cost$ auf den Wert $a$}
**function** $search(x.pos)$: Pointer;
{sucht das unter dem Suchschlüssel $x.pos$ abgelegte Element und liefert ggf. einen Zeiger darauf zurück}

Die Liste *OPEN* unterstützt die Operationen einer *Vorwärtswarteschlange*, nämlich $is\_empty$, $insert$, $access\_min$ (liefert hier das bzgl. dem $f$-Wert minimale Element), $delete\_min$, und $decrease\_key$. Darüberhinaus muß sie aber auch die Operation $search$ unterstützen. Eine konkrete Realisierung läßt sich nicht allein durch eine Vorwärtswarteschlange realisieren, da diese die Suchoperation nicht unterstützt. Die Vorwärtswarteschlange kann in Verbindung mit *Hashing* implementiert werden, was eine effiziente Implementierung der *search* Operation darstellt.

Für *CLOSED*:

**procedure** $initiate()$;

{initialisiert die Datenstruktur *CLOSED*}
**procedure** $insert(x.pos)$;
{fügt den Knoten unter dem Suchschlüssel $x.pos$ ein}
**function** $search(x.pos)$: Pointer;
{sucht das unter dem Suchschlüssel $x.pos$ abgelegte Element}
**procedure** $delete(x.pos)$;
{entfernt das Element $x$ mit Suchschlüssel $x.pos$}

*CLOSED* muß die üblichen *dictionary* Operationen, wie *insert*, *search* und *delete* bereitstellen, was ebenfalls durch ein Hashverfahren effizient realisiert werden kann.

Nun folgt eine Beschreibung des A*-Algorithmus in Pascal-ähnlicher Notation. Wie es in objektorientierten Programmiersprachen üblich ist, stellen wir den Namen der Datenstruktur dem Funktions-/Prozeduraufruf voran.

**Algorithm** A*;

**begin**
*OPEN*.initiate;
*CLOSED*.initiate;
$s.pos :=$ Startkonfiguration;
$s.cost := h(s.pos)$; {d. h. $g(s.pos) := 0$}
$s.pred := nil$;
*OPEN*.insert($s.cost$, $s.pos$);
**while not** *OPEN*.is_empty **do**
$x :=$ OPEN.$accessmin$;
*OPEN.deletemin*;
*CLOSED.insert*($x.pos$):
**if** $goal(x.pos)$ **then** {$x.pos$ ist eine Zielkonfiguration}
**exitwhile**; {Suche erfolgreich beendet}
$\Gamma(x.pos) := expand(x.pos)$; {expandiere den Knoten $x$,
d. h. erzeuge die, evtl. leere, Menge der Nachfolgerkonfigurationen $\Gamma(x.pos)$ von $x.pos$ in G}
**forall** $x'.pos \in \Gamma(x.pos)$ **do**
$x'.cost := x.cost + (h(x'.pos) + c(x.pos, x'.pos) - h(x.pos))$;

$\{$da $f(x') = g(x') + h(x') = g(x) + c(x,x') + h(x')$
$= f(x) - h(x) + c(x,x') + h(x')\}$
$x'.pred := address\_of(x)$; $\{address\_of$ sei der Adressoperator$\}$
$p_1 := OPEN.search(x'.pos)$;
**if** $p_1 \neq nil$ **then** $\{x'.pos \in OPEN$ and $x'.pos \notin CLOSED\}$
**if** $x'.cost < p_1 \uparrow .cost$ **then** $\{x'.pos$ ist nun
über einen günstigeren Weg erreicht worden$\}$
$p_1 \uparrow .pred := address\_of(x)$;
$OPEN.decrease\_key(p_1, x'.cost)$;
**endif**
**else**
$p_2 := CLOSED.search(x'.pos)$;
**if** $p_2 \neq nil$ **then** $\{x'.pos \notin OPEN$ und $x'.pos \in CLOSED\}$
**if** $x'.cost < p_2 \uparrow .cost$ **then** $\{x'.pos$ ist nun
über einen günstigeren Weg erreicht worden$\}$
$CLOSED.$delete$(p_2 \uparrow .pos)$;
$OPEN.insert(x'.cost, x'.pos)$;
**endif**
**else** $\{x'.pos \notin OPEN$ and $x'.pos \notin CLOSED\}$
$OPEN.$insert$(x'.cost, x'.pos)$;
**endif**
**endif**
**endfor**
**endwhile**

Der Funktionsaufruf $expand(x)$ erzeugt zu jedem Nachfolger von $y \in \Gamma(x.pos)$ in $G$ ein Element $x'$ vom Typ Knoten und initialisiert $x'$ durch $x'.pos = y$, $x'.cost = 0$ und $x'.pred = NIL$. In einer konkreten Implementation kann es vorteilhaft sein, zu einem Nachfolger $x'$ die Kosten der Kante $(x.pos, x'.pos)$ in $x'.cost$ zurückzuliefern. Noch zweckmäßiger ist es vielleicht $x'.cost$ gleich mit dem Wert $f(x')$ zu belegen, denn wegen

$$\begin{aligned} f(x'.pos) &= g(x'.pos) + h(x'.pos) \\ &= g(x.pos) + c(x.pos, x'.pos) + h(x'.pos) \\ &= f(x.pos) + c(x.pos, x'.pos) + h(x'.pos) - h(x.pos) \end{aligned}$$

berechnet sich $f(x')$ als Summe von $f(x)$ und spiel-(problem)spezifischen gegebenen Werten. Der Wert der Differenz $h(x'.pos) - h(x.pos)$ läßt sich unter Umständen effizienter bestimmen, als die Werte $h(x'.pos)$ und $h(x.pos)$. Zum Beispiel beim Fünfzehnerpuzzle mit der Manhattan-Distanz als Heuristik läßt sich obige Differenz allein aus den durch einen Zug veränderten, benachbarten Spielfeldern berechnen. Die Belegung der anderen Felder benötigt man dabei nicht.

In einer Implementierung kann es zudem von Vorteil sein die Elemente der *OPEN* und *CLOSED*-Liste in einer gemeinsamen Suchstruktur einzufügen. Mit einem Bit läßt sich dann kenntlich machen, ob ein gegebenes Element aus der *OPEN* oder *CLOSED*-Liste ist. In diesem Fall müssen keine Elemente aus der Suchstruktur entfernt werden, denn durch ein Kippen des Bits wird ein Element von der *OPEN*-Liste in die *CLOSED*-Liste übertragen.

Im folgenden wollen wir aus Gründen der Übersichtlichkeit und besseren Lesbarkeit von einer zu detaillierten, technisch orientierten Darstellung absehen. So nehmen wir nun auch generell an, daß die die Elemente von $\Gamma(x)$ bzw. von $S(x) \subset \Gamma(x)$ bereits wie beim A*-Verfahren initialisiert werden.

## 3.3 Das Verfahren IDA*

IDA* ist eine iterierte Tiefensuche. In jedem IDA*-Schritt wird eine Tiefensuche durchgeführt, beginnend von der Wurzel des Suchbaumes, die an einem Knoten genau dann abgebrochen wird, wenn der $f$-Wert dieses Knotens einen vorgegebenen Schwellwert (engl.: threshold) $t$ überschreitet. Als Schwellwert des folgenden IDA*-Schrittes wird der minimale $f$-Wert aller im vorangegangenen Schritt ermittelteten Horizontknoten gewählt. Initialisiert wird dies durch die Expansion der Wurzel im ersten Schritt. In einem IDA*-Schritt werden alle $t$-beschränkten Knoten expandiert. IDA* ist $f^*$-geordnet, da immer alle Knoten, deren $f^*$-Wert kleiner gleich $t$ ist, expandiert werden, d.h. daß ein Knoten dessen $f^*$-Wert sich erhöht erzeugt, aber nicht expandiert wird. Der Hauptvorteil gegenüber best-first Suchverfahren besteht darin, daß der Speicherbedarf sehr gering ist, da IDA* nur den aktuellen Suchpfad speichert. Wir geben eine rekursive algorithmische Beschreibung.

```
Algorithm IDA*;
begin
    s := Startkonfiguration;
    t := h(s);
(1) while ein Zielknoten ist nicht erreicht and t ≠ ∞ do
        t := IDA*_generate(s, t);
    endwhile
end


function IDA*_generate(x: Knoten, t: integer) : integer
    {t ist der aktuelle Schwellwert, d. h. der Schwellwert für diesen
    Durchlauf.
    Das Ergebnis ist der Schwellwert für den nächsten Durchlauf.}
var
    t^(new): integer;

begin
    if f(x) > t then {die Tiefensuche wird hier
                      an einem Horizontknoten abgebrochen}
        return(x.cost);
    else
        if x ist ein Zielknoten then exit;
        else
            t^(new) := ∞;
            Γ(x) := expand(x);
            foreach x' ∈ Γ(x) do
                t^(new) := min{IDA*_generate(x', t),t^(new)};
                if ein Zielknoten ist gefunden then exit;
            endfor
        endif
        return(t^(new));
end
```

Eine Iteration der *while*-Schleife in (1) bezeichnen wir als *Schritt*. IDA$^*$ bricht ab, falls eine Lösung gefunden wird, oder falls der Graph vollständig

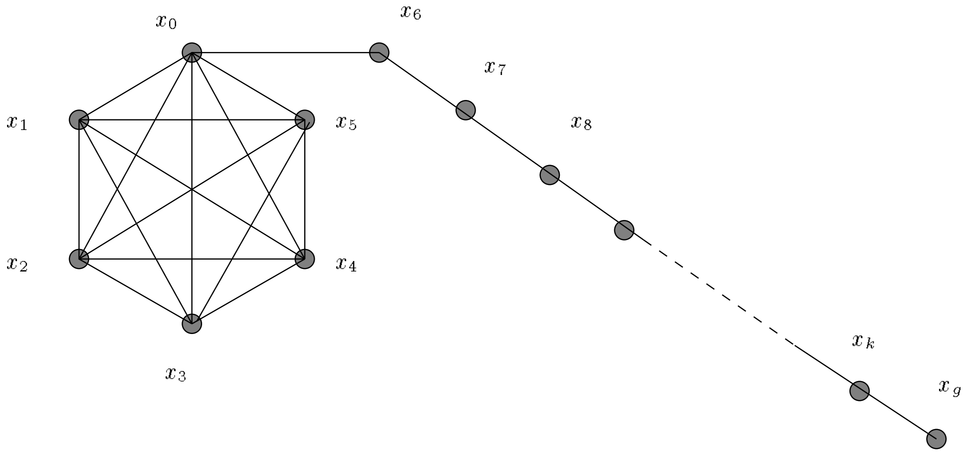


Abbildung 1:

expandiert worden ist (d. h. $t = \infty$). Bricht IDA* mit einer Lösung ab, so ist die gefundene Lösung optimal, d. h. IDA* ist zulässig. Der Lösungspfad läßt sich aus den auf dem Stack gespeicherten Knoten der rekursiv aufgerufenen IDA*_generate Prozedur rekonstruieren. In jedem Schritt expandiert IDA* alle Knoten $x$ mit $f^*(x) \leq t$. Das bedeutet, daß wiederum alle Knoten des vorangegangenen Schrittes expandiert werden.

Da im Gegensatz zu A* Duplikate mehrmals erzeugt und expandiert werden, kann selbst im Fall von Graphen mit uniformen Kosten der Aufwand von IDA* exponentiell gegenüber dem Aufwand von A* wachsen. Siehe dazu das folgende Beispiel.

**Beispiel:** Der Graph $G_k, k \geq b$, mit den Knoten $\{x_1, x_2, \ldots, x_k, x_g\}$ sei wie in Abbildung 1 definiert: Die Menge $\{x_0, x_1, \ldots, x_5\}$ spanne einen vollständigen Teilgraphen mit dem Verzweigungsgrad 5 auf. Der Knoten $x_g$ sei der Zielknoten.

A* expandiert diese Menge genau einmal, und damit insgesamt genau $k$ Knoten. IDA* dagegen expandiert die Knoten aus $B$ immer wieder von neuem, d. h. $N_{\mathrm{IDA}^*}(G_k) > 5^k$.

Für $c$-beschränkte Graphen gilt jedoch folgendes Lemma.

**Lemma 3.1** *Sei $c \in I\!R^+$ und $G$ ein $c$-beschränkter Graph. Dann gilt für $G$: $N_{\mathrm{IDA}^*} \leq \frac{c}{2}(N_{\mathrm{A}^*})^2$; d.h. $N_{\mathrm{IDA}^*} = \Theta(N^2_{\mathrm{A}^*})$ auf der Menge der $c$-beschränkten Graphen.*

**Beweis:** Der für IDA$^*$ ungünstigste Fall liegt dann vor, wenn in jedem neuen Schritt nur ein Knoten expandiert wird. Setzen wir $n = N_{\mathrm{A}^*}$, so folgt

$$N_{\mathrm{IDA}^*} = \sum_{i=1}^{n} ci = c\frac{n(n+1)}{2} \in O(cn^2)$$

□

Wir definieren einige Varianten der Algorithmen A$^*$, IDA$^*$, die man durch geringfügige Änderungen der klassischen Verfahren erhält, und im folgenden als Grundbausteine für den Aufbau komplizierterer Algorithmen benötigt werden.

**function** A$^*$*_limit*($k$ **: integer) : Datenstruktur:**

Diese Funktion führt A$^*$ solange durch, bis insgesamt $k$ viele Knoten gespeichert worden sind. Sie liefert die Menge der Horizontknoten in einer Datenstruktur zurück.

**function** IDA$^*$*_next_thresh*($x$ **: Knoten,** $t$ **: integer) : integer:**

Diese Funktion führt einen IDA$^*$-Schritt durch, und zwar vom Startknoten $x$ mit dem Schwellwert $t$. Sie liefert als Ergebnis das Minimum aller $f$-Werte der Horizontknoten des erzeugten Teilbaumes.

**function** IDA$^*$*_size_of_tree*($x$ **: Knoten,** $t$ **: integer) : integer:**

Diese Funktion führt ebenfalls einen IDA$^*$-Schritt durch, und zwar vom Startknoten $x$ mit dem Schwellwert $t$. Sie liefert als Ergebnis die Anzahl der Knoten des erzeugten Teilbaumes.

Soll sowohl die Größe als auch das Minimum der $f$-Werte des erzeugten Teilbaums bestimmt werden, so ist es möglich mit einer IDA$^*$-Expansion des Baumes beide Werte zugleich zu bestimmen. Im folgenden werden wir jedoch die Funktionen IDA$^*$*_next_thresh* und IDA$^*$*_size_of_tree* als von einander unabhängig betrachten, um die unterschiedlichen Ergebnisse deutlicher zu trennen.

# 4 Speicherplatzbeschränkte Algorithmen

In diesem Abschnitt stellen wir die bisher bekannten Algorithmen vor, die unter beschränktem Speicherplatz arbeiten.

## 4.1 Die Algorithmen IDA*_CR, MIDA* und DFS*

Da IDA* in einem Schritt $i$ den Schwellwert $t$, mit $t := min\{f(x)|x \in \textsc{open}\}$, nimmt, wird evtl. nur ein weiterer Knoten expandiert. Natürlich kann man durch die Wahl eines höheren Schwellwertes im Schritt $i$ eine höhere Anzahl expandierter Knoten garantieren. Diese Idee führt zu dem Verfahren IDA*_CR [2]. Seien $f_1^{(i)}, f_2^{(i)}, \ldots, f_{n_i}^{(i)}$ die $f$-Werte der Horizontknoten des $i$-ten Schrittes, so wird in [2] vorgeschlagen, als Schwellwert $t := f_{b^i}^{(i)}, b \geq 2$, für einen neuen Schritt $i+1$ zu setzen. Weiterhin wird empfohlen für $b$ eine Abschätzung des effektiven Verzweigungsgrades zu verwenden. Bei der Anwendung auf Travelling-Salesman Probleme hat sich diese Wahl bewährt und zu einer erheblichen Reduktion der Knotenexpansionen geführt.

Statt wie oben $t := f_{b^i}$ zu wählen, ist es auch denkbar gleich $t := f_{n_i}^{(i)}$ zu setzen. Keine dieser IDA*-Varianten liefert jedoch einen geordneten Algorithmus. Ein Algorithmus, der nicht geordnet ist, besitzt eine Komplexität von $\Omega(n^2)$. Weshalb das so ist, können wir anhand von IDA*_CR für die letzte Wahl des Schwellwertes $t$ an folgendem Beispiel sehen.

**Beispiel:** Wir definieren eine Klasse von Graphen $G_k, k \geq 1$, wie in Abbildung 2 gezeigt.

Der Graph $G_k$ sei nahezu uniform, d. h. alle Kanten, mit Ausnahme $(p(y), y)$, haben das Gewicht 1. Das Gewicht der Kante $(p(y), y)$ sei $k^3$. Nehmen wir wieterhin an, daß die Heuristik den Wert 0 liefere, also der $f$-Wert eines Knoten $x$ durch das Gewicht des Pfades von der Wurzel zum Knoten $x$ gegeben ist. Dann ist klar, daß A* nur Knoten des oberen Teilbaumes expandiert, also $N_{\mathrm{A}^*}(G_k) < b^k$ ist. IDA*_CR expandiert die Knoten des Levels $i, i < k$, in einem Schritt. Mit dem nächsten Schwellwert wird jedoch auch der untere Teilbaum vollständig expandiert, woraus $N_{\mathrm{IDA^*\_CR}} > b^{3k-1}$ folgt. Damit erhalten wir $N_{\mathrm{IDA^*\_CR}} > N_{\mathrm{A}^*}^2$.

Es ist klar, daß sich auf diese Weise beliebig schlechte Beispiele konstruieren lassen. Auch in praktischen Beispielen kann der Aufwand im letzten Schritt erheblich größer sein, als in allen vorangegangenen. Es gilt genauer:

**Lemma 4.1** *Für alle $c$-beschränkten Graphen $G$ ist $N_{\mathrm{IDA^*\_CR}} \leq 2cN_{\mathrm{A}^*} + n_{last}$, wo $n_{last}$ die Anzahl der von* IDA* *im letzten Schritt expandierten*

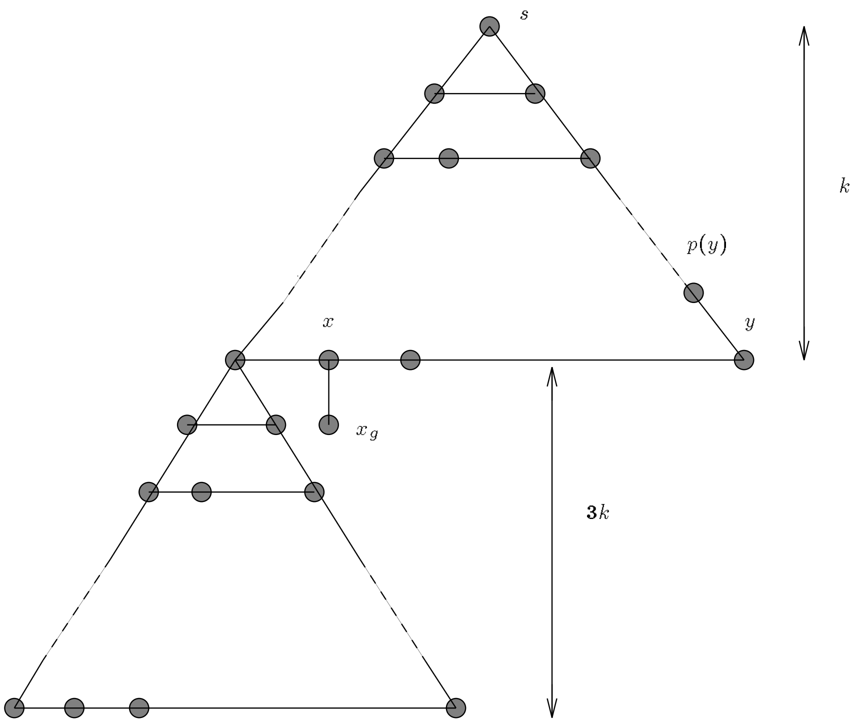


Abbildung 2:

*Knoten sind.*

**Beweis:** Nicht ausgeführt. □

## 4.2 RBFS, IE

RBFS [4] bzw. SRBFS [4] sind rekursive Algorithmen, die, ähnlich wie IDA*, nur den aktuellen Suchpfad und die Söhne der Knoten dieses Pfades speichern. Die Expansion der nächsten Horizontknotens wird jedoch nicht immer wieder erneut von der Wurzel aus gestartet, sondern von demjenigen Knoten $x$ des aktuell gespeicherten Pfades, der diesen Horizontknoten enthält. Dazu wird beim Aufruf eines IDA*-Schrittes vom Knoten $x$ der minimale $f$-Wert aller anderen Horizontknoten, die nicht Nachfolger von $x$ sind, als Schranke übergeben. Die folgenden Expansionen von Horizontknoten erfolgen solange unter den Nachfolgern von $x$, wie der minimale $f$-Wert aller Horizontknoten des Teilbaumes mit Wurzel $x$, kleiner oder gleich der Schranke ist. Ist das nicht mehr der Fall, wird die weitere Expansion in diesem Teilbaum abgebrochen, und der Algorithmus geht in der Rekursion wieder eine Stufe zurück. Bei Experimenten mit dem 15-Puzzle expandierte RBFS allerdings eine geringfügig höhere Anzahl von Knoten als IDA*. Das liegt an Unregelmäßigkeiten im letzten Iterationsschritt bei einer zufällig gewählten *tie-breaking* Regel. Die Worst-case Komplexität ist jedoch von der gleichen Größenordnung wie bei IDA*. Der Algorithmus IE [10] ist ähnlich zu RBFS.

## 4.3 MREC

Dieser Algorithmus führt A* durch bis der zur Verfügung stehende Speicherplatz fast vollständig gefüllt ist. Anschließend wird IDA* ausgeführt. Duplikate werden erkannt, sofern neu generierte Knoten unter den bereits gespeicherten Knoten auftreten. Einmal mittels A* erzeugte und gespeicherte Knoten werden zu einem späteren Zeitpunkt nicht mehr entfernt. Eine dynamische Anpassung des gespeicherten Graphen an den Horizont des aktuellen Suchbaumes wird nicht vorgenommen. Da aber der Suchbaum nicht gleichmäßig in die Tiefe entwickelt wird, sondern, dank der

Heuristik, dort, wo mit höherer Wahrscheinlichkeit der Zielknoten zu finden ist, ist dieses völlig statische Verfahren unbefriedigend. Wir werden später sehen, daß sich sowohl im Worst-case als auch im allgemeinen Fall die Anzahl expandierter Knoten reduzieren läßt, wenn der gespeicherte Graph dynamisch dem Horizont des Suchbaumes angepaßt wird.

## 4.4 MA*, SMA* und ITS

Während MREC den Speicherplatz mit den ersten der generierten Knoten statisch belegt, belegt MA* den Speicherplatz dynamisch, in dem unter den bisher erzeugten Knoten, diejenigen mit kleinstem $f$-Wert und deren Vorgängerknoten gespeichert werden [10].

Zunächst verhält sich MA* wie A* bis der Speicherplatz fast vollständig belegt ist. Dann werden, um neuen Speicherplatz zu gewinnen, bereits gespeicherte Knoten wieder entfernt (in der Original- Version sogar alle mit Ausnahme der Knoten des bisher günstigsten Blatt-Wurzel-Pfades). Anschließend wird der Speicherplatz wieder mit den Knoten in der Reihenfolge ihrer Erzeugung belegt. Das führt bei höheren Suchtiefen dazu, daß MA* nur die Knoten in der Umgebung der zuletzt expandierten Horizontknoten und die Knoten des besten Blatt-Wurzel-Weges speichert.

Dieses Vorgehen hat einen gravierenden Nachteil. Die Knoten werden bevorzugt im unteren Teil des Suchbaumes in der Umgebung der günstigsten Blatt-Wurzel-Wege gespeichert. Duplikate werden somit im oberen Teil des Suchbaumes nicht erkannt. Wird ein Knoten der *OPEN*-Liste außerhalb dieses Bereichs expandiert, so werden letztlich alle bisher gespeicherten Knoten wieder gelöscht. Treten zwei Duplikate, sagen wir $x$ und $y$, im oberen Teil des Suchbaumes auf, deren Teilbäume nicht mehr gespeichert werden können, so werden deren Teilbäume, wenn einer ihrer Horizontknoten expandiert wird, wechselweise immer wieder gänzlich neu erzeugt und abgespeichert. Das führt zu einem hohen Update Aufwand, der das Verfahren sehr ineffizient macht. Dieser Fall ist in der Praxis durchaus keine Seltenheit, sondern z. B. typisch für das Fünfzehnerpuzzle, wo ja viele auch sehr unterschiedliche Zugfolgen wieder die gleiche Stellungen liefern können. Selbst wenn dieser Extremfall nicht eintritt, so müssen für die Expansion eines einzigen Horizontknotens viele Update Operationen, wie das Einfügen und Entfernen von Knoten in einen bzw.

aus einem Heap, durchgeführt werden.

Im Hinblick auf die Komplexität des Verfahrens, ist es offenbar ratsam, die Knoten im oberen Teil des Suchbaumes zu speichern, da das Erkennen von Duplikaten hier größere Einsparungen verspricht. Selbst für den Fall, daß die Graphen Bäume sind, ist es zweifelhaft, ob die von MA* vorgenommene Verteilung von gespeicherten Knoten besonders vorteilhaft ist.

Die Komplexität von MA* wird lediglich für den Fall von vollständigen Bäumen mit uniformen Kosten betrachtet [10]. Hierbei weist das Verfahren eine Komplexität von $O(N_{\mathrm{A}^*})$ auf. Das ist jedoch der günstigste Fall. In anderen Fällen, selbst für beliebige 1-beschränkte Graphen, d.h. für Graphen, in denen es zu jedem Knoten nur einen einzigen Weg von $s$ gibt, ergibt sich wieder eine Komplexität von $\Omega((N_{\mathrm{A}^*})^2)$.

Der angesprochene Nachteil der in hohem Maße für MA* gilt, gilt zumindest tendenziell auch für SMA*. SMA* stellt insoweit eine Verbesserung gegenüber MA* dar, als hier immer nur ein Knoten entfernt wird, aber ebenfalls vorzugsweise ein Knoten mit geringerer Tiefe. Ein weitere Verbesserung besteht darin, daß dabei die Information über den minimalen $f$-Wert der Horizontknoten des entfernten Teilbaumes gerettet wird, d. h. jeder Horizontknoten des expliziten Graphen speichert den minimalen $f$-Wert aller bisher erzeugten Nachfolger. Anzumerken ist noch, daß ausschließlich Horizontknoten expandiert werden, die Nachfolger von Blättern des expliziten Graphen sind. Dies stellt ebenfalls eine eher nachteilige Einschränkung für die Wahl des nächsten zu expandierenden Knotens dar.

Der prinzipielle Nachteil der Verfahren, der sie für praktische Anwendungen ungeeignet macht, liegt jedoch in dem hohen Overhead, bedingt durch die hohe Zahl von Update-Operationen in der *OPEN*- und *CLOSED*-Liste. Jede neue Erzeugung eines Knotens führt zu einer Einfüge- und ggf. auch einer oder mehreren Entferne-Operationen sowohl in der *OPEN* als auch in der *CLOSED*-Liste. Die Anzahl der ausgeführten Update-Operationen ist also von derselben Größenordnung, wie Anzahl der Knotenexpansionen. Da die *OPEN*-Liste als Vorwärtswarteschlange und die *CLOSED*-Liste als Wörterbuch implementiert werden, sind solche Update-Operationen i. a. wesentlich teurer als z. B. Stapel-Operationen, wie sie das IDA*-Verfahren ausschließlich benötigt. Wir werden Algorithmen präsentieren, die bei den meisten Knotenexpansionen solche teuren Update-Operationen vermeiden.

# 5 Neue speicherplatzbeschränkte Algorithmen

Wir stellen im folgenden mehrere Algorithmen vor, die eine Zeitkomplexität von $O(N^2_{\mathrm{A}^*}/m)$ bzw. $O(dN^2_{\mathrm{A}^*}/m)$ besitzen, wobei $d$ die maximale Suchtiefe und $m$ die Größe des Speicherplatzes ist. Wir setzen hier voraus, daß zudem ein Speicherplatz der Größe $d$ zum Speichern des aktuellen Suchpfades zur Verfügung steht. Die dabei zur Anwendung kommenden Methoden lassen sich mühelos auf die oben genannten Algorithmen übertragen. Die Algorithmen werden zunächst als Baum-Suchalgorithmen vorgestellt und analysiert. Die Ergebnisse der Analyse gelten auch für $c$-beschränkte Graphen sofern $c$ konstant ist. Wie die Baum-Suchalgorithmen zu Graph-Suchalgorithmen modifiziert werden können, wird im Abschnitt 5.5 erläutert.

## 5.1 Der Algorithmus MEIDA*

Der Algorithmus MEIDA* (Memory Extended IDA*) ist stark an den IDA*-Algorithmus angelehnt und besteht aus zwei einander abwechselnden Phasen. In der ersten Phase werden zunächst, ähnlich wie bei IDA*, alle Knoten expandiert, deren $f^*$-Wert kleiner gleich einem vorgegebenen Schwellwert ist. Im Gegensatz zu IDA* wird jedoch nicht nur das Minimum der $f^*$-Werte aller Horizontknoten bestimmt, sondern es werden die $m$ Horizontknoten $x_1, x_2, \ldots, x_m$ mit kleinstem $f^*$-Wert in einer Vorwärtswarteschlange $D$ (entsprechend ihrem $f^*$-Wert) eingefügt. Wie bereits erwähnt gilt für alle im IDA*-Schritt erzeugten Horizontknoten $x$, daß $f^*(x) = f(x)$ ist und daher $f^*(x)$ leicht berechnet werden kann.

In der zweiten Phase werden diese $m$ Knoten expandiert. Hierzu wird immer wieder das Minimum von $D$ expandiert und aus $D$ entfernt, solange bis $D$ wieder leer ist. Um zu garantieren, daß nur Horizontknoten mit minimalem $f^*$-Wert expandiert werden, ist es erforderlich, daß auch die Söhne expandierter Knoten in $D$ eingefügt werden, wenn deren $f^*$-Wert kleiner als der maximale $f^*$-Wert in $D$ ist. Da in $D$ nicht mehr als $m$ viele Elemente gespeichert werden können, muß gegebenenfalls mit dem Einfügen eines neuen Knotens das maximale Elemente aus $D$ entfernt werden. Sobald $D$ leer ist, wird wieder ein IDA*-Schritt gestartet mit dem $f^*$-Wert des zuletzt expandierten Knotens als Schwellwert.

Wir setzen voraus, daß $D$ sowohl die Operationen eines Min-Heaps als auch eines Max-Heaps unterstützt, d. h. insbesondere die Operationen *access_min*, *delete_min*, *access_max* und *delete_max*. Unterstützt $D$ zudem noch die Operation *search*, so lassen sich Duplikate in $D$ vermeiden. Dies läßt sich durch die Kombination eines Min- bzw. Max-Heaps mit einem Hashverfahren (für die Suchoperation) realisieren.

Es folgt eine Beschreibung des Algorithmus in Pascal-ähnlicher Notation.

**Algorithmus** MEIDA*;
**begin**
{Initialisierung der folgenden Schleife}
$s$ := Startkonfiguration;
$t^{(new)} := h(s)$;
$t := 0$;
**while** $t \neq \infty$ **do**

{Phase 1}
$t := t^{(new)}$;
$(D, t^{(new)})$ := IDA*_*next_thresh*$(s, t)$;
{speichere die $m$ Horizontknoten mit kleinstem $f^*$-Wert in $D$ ab und lege den neuen Schwellwert $t^{(new)}$ fest}

{Phase 2}
**while** $D$ ist nicht leer **and** ein Zielknoten ist nicht erreicht **do**
$x := \min_{f^*}(D)$;
$D$.*delete_min*;
**if** $x$ ist ein Zielknoten **then exit**;
(1) $S(x) := expand(x)$;
(2) **foreach** $x' \in S(x)$ **do**
$f^*(x') := \max\{f(x'), f^*(x)\}$

$y := \max_{f^*}(D)$;
**if** $f^*(x') \leq f^*(y)$ **then** $D.insert(x', f^*(x'))$;
**if** $|D| > m$ **then** $D$.delete_max; $\{D$ enthält nie mehr als $m+1$ viele Elemente$\}$
**endfor**
$t := f^*(x)$; { $t$ ist der neue Schwellwert }

**endwhile**
**endwhile**
**end**

Falls die Heuristik nicht monoton ist, können in Schritt (1) Knoten erzeugt werden, die einen kleineren $f$-Wert als ihre Vorgänger haben, d.h. Knoten $x$ für die $f(x) < f^*(x)$ ist. Da $h$ optimistisch ist, gilt jedoch für einen Zielknoten $x^*$ immer $f(x^*) = f^*(x^*)$ (andernfalls gäbe es einen Knoten $x \in path(x^*)$ für den $h$ nicht optimistisch ist). Daher können Knoten mit $f(x) < f^*(x)$ keine Zielknoten sein und müssen auch nicht in $D$ eingefügt werden. Somit ist es sinnvoll, Schritt (1) durch IDA*_*next_thresh*$(x, f^*(x))$ zu ersetzen und die Foreach-Schleife (2) auf alle in diesem IDA*-Schritt erzeugten Horizontknoten anzuwenden, ohne die von IDA*_*next_thresh* erzeugten Knoten in $D$ zu verwalten.

Wir zeigen im folgenden, daß MEIDA* zulässig ist und nur $O(N_{\mathrm{A}^*}^2/m)$ viele Knoten expandiert. Für den Beweis der Zulässigkeit greifen wir auf Satz 2.2 zurück.

**Satz 5.1** *Der Algorithmus* MEIDA* *ist zulässig.*

**Beweis:** Da wir nur Probleminstanzen in $I_{AD}$ betrachten, genügt es nach Satz 2.2 zu zeigen, daß MEIDA* geordnet ist. Der Beweis erfolgt durch Induktion über die Anzahl der Durchläufe der inneren WHILE-Schleife (und damit über die Anzahl von Durchgängen der Phase 1 und 2).

In der ersten Ausführung von Phase 1 werden nur Knoten expandiert, die den gleich $f^*$-Wert wie $s$ haben. Daher gilt für alle in $D$ gespeicherten Knoten $x$:

1. $x$ ist nicht expandiert,

2. $f^*(x) > t = h(s)$,

3. $f^*(x)$ ist kleiner als alle $f^*$-Werte von erzeugten, aber nicht expandierten Knoten, die nicht in $D$ gespeichert worden sind, und

4. $t$ ist der größte $f^*$-Wert bereits expandierter Knoten.

5. Knoten, deren $f^*$-Wert kleiner $t$ ist, wurden bereits expandiert (oder ein Duplikat davon). Bei der ersten Ausführung der Phase 1 gibt es trivialerweise keine solche Knoten.

Wie man leicht sieht, sind die obigen Bedingungen die Invariante der inneren While-Schleife. Dies folgt aus der Tatsache, daß in Phase 2 immer nur das Minimum von $D$ expandiert wird. Daher expandiert MEIDA* in der ersten Iteration der äußeren While-Schleife alle Knoten in der $f^*$-Ordnung. Für die folgenden Iterationen gelten die obigen Überlegungen entsprechend. □

**Satz 5.2** *Der Algorithmus* MEIDA* *expandiert* $O(N_{\mathrm{A}^*}^2/m)$ *viele Knoten unter Ausnutzung eines Speicherplatzes der Größe* $m$.

**Beweis:** Sei $I$ eine Probleminstanz in $I_{CON}$ und $n := N_{\mathrm{A}^*}(I)$, d. h. die Anzahl der Knotenexpansionen von A* auf $I$. Wir bezeichnen den expandierten Graphen nach dem $i$-ten Durchlauf der äußeren While-Schleife mit $T^{(i)}, 0 \leq i \leq l$ ($T^{(0)}$ ist der leere Baum). Sei $S^{(i)} = T^{(i)} - T^{(i-1)}$, $t_i = |T^{(i)}|$ und $s_i = |S^{(i)}|$, für $1 \leq i \leq l$, wobei $l$ die letzte Iteration sei (in der der Zielknoten gefunden worden wird).

Also gilt $N_{\mathrm{MEIDA}^*}(I) = \sum_{i=1}^{l} t_i$. Da $t_i = \sum_{j=1}^{i} s_i$ ist, ergibt sich $N_{\mathrm{MEIDA}^*}(I) = \sum_{i=1}^{l} i s_{l-i+1}$. In MEIDA* wird sichergestellt, daß $s_i \geq m$, für $2 \leq i < l$, und $s_l \geq 0$ im letzten Schritt ist. Durch eine geringfügige Modifikation läßt sich auch für den ersten Schritt $s_1 \geq m$ erzielen. Mit $n = |T^{(l)}|$ folgt, daß sich eine obere Schranke für die maximale Anzahl von Knotenexpansionen von MEIDA* als Lösung der folgenden Optimierungsaufgabe ergibt:

$$\max \sum_{i=1}^{l} (l + 1 - i) s_i$$

Unter den Nebenbedingungen:

$$\sum_{j=1}^{l} s_i = n$$
$$s_i \geq m, \text{ für } 1 \leq i < l, s_l \geq 0.$$

Wie leicht zu sehen ist, ergibt sich für festes $n$ und $l$ die optimale Lösung als $s_l = 0$, $s_i = m$, für $1 < i < l$, und $s_1 = n - (l-2)m$, was zu einem Zielfunktionswert von

$$\sum_{i=2}^{l-1} im + l(n - m(l-2)) = -\frac{m}{2}l^2 + (n + \frac{3m}{2})l - m$$

führt. Das Maximum des Zielfunktionswertes wird für $l = (n + 3m/2)/m$ angenommen und ergibt einen Wert von $n^2/(2m) + 3n/2 + m/8 \in O(n^2/m)$ wie behauptet. □

## 5.2 Statische Baumbalancierung

Die Idee der statischen Baumbalancierung besteht darin, daß man im Gegensatz zu Algoritmus MEIDA*, der ausschließlich Knoten der Horizontlinie speichert, die zu speichernden Knoten so über den bereits erzeugten Suchbaum verteilt, daß alle von diesen Knoten erzeugten Teilbäume ungefähr gleich groß sind. Diese Idee wird in Algorithmus SBIDA* (Static Balanced IDA*) verwirklicht.

Der Algorithmus SBIDA* verläuft zunächst analog zu A*. Belegt jedoch im Schritt $i_0$ der erzeugte Baum $T^{(i_0)}$ den gesamten zur Verfügung stehenden Speicherplatz, so lassen sich für $i > i_0$ nicht mehr alle im $i$-ten Schritt erzeugten Knoten abspeichern. Stattdessen kann nur ein Teilgraph $U^{(i)}$ von $T^{(i)}$ gespeichert werden. Die Knoten von $U^{(i)}$ bezeichnen wir im folgenden auch als *Kontrollknoten.* Wir sagen, ein Knoten $x$ in $T^{(i)}$ *kontrolliert* einen Nachfolger $y$, falls auf dem Pfad von $x$ nach $y$ kein Kontrollknoten liegt. Wenn wir annehmen, daß $s$ in $U^{(i)}$ enthalten ist, so gibt es zu jedem Knoten $y$ in $T^{(i)}$ einen Knoten $x$ in $U^{(i)}$, der $y$ kontrolliert. Im folgenden zeigen wir, wie der Teilgraph $U^{(i)}$ verwendet werden kann, um die Anzahl der Knotenexpansionen zu reduzieren.

Um im $i$-ten Schritt den Horizontknoten $y$ von $T^{(i)}$ mit minimalem $f$-Wert (bzw. $f^*$-Wert) zu expandieren, wird IDA* gestartet, jedoch nicht beginnend bei der Wurzel $s$, sondern von demjenigen Knoten $x$ in $U^{(i)}$, der $y$ kontrolliert. Der Schwellwert des IDA*-Schrittes ist dabei natürlich gleich dem $f$-Wert von $y$. Der minimale $f$-Wert aller in diesem IDA*-Schritt erzeugten Horizontknoten wird bei $x$ gespeichert. Im folgenden bezeichnen wir diesen Wert mit $f_x^{(i)}$ (oder kurz $f_x$). Die Invariante des Algorithmus lautet dann, daß der Horizontknoten $y$ mit dem kleinsten $f$-Wert von demjenigen Knoten $x \in U^{(i)}$ kontrolliert wird, der den kleinsten $f_x$-Wert besitzt.

Alle Knoten $x$ von $U^{(i)}$ werden in einer Vorwärtswarteschlange $D$ unter Verwendung ihres $f_x$-Wertes als Schlüssel gespeichert. Der IDA*-Schritt wird immer von dem Minimum $x$ von $D$ mit Schwellwert $f_x$ gestartet. Anschließend wird $D$ aktualisiert, indem $x$ entsprechend dem neuen Wert $f_x$ in $D$ eingefügt wird.

Dabei entsteht das Problem, daß in den einzelnen Teilbäumen die Horizontknoten mit unterschiedlicher Häufigkeit expandiert werden und so die Teilbäume unterschiedlich schnell wachsen. Um die Anzahl der Knotenexpansionen möglichst gering zu halten, ist es aber erforderlich, daß die maximale Anzahl der von einen Knoten in $U^{(i)}$ kontrollierten Knoten möglichst gering ist. Hierzu führen wir den Begriff der Kontrollgröße eines Knotens ein. Die *Kontrollgröße* $n_x^{(i)}$ eines Knotens $x$ im Schritt $i$ ist die Anzahl der von $x$ im Schritt $i$ kontrollierten Knoten. Wie man leicht sieht, wird das Maximum der Kontrollgrößen der Knoten von $U^{(i)}$ genau dann minimiert, wenn jeder Knoten in $U^{(i)}$ ungefähr die gleiche Kontrollgröße hat. Dies entspricht der Forderung, daß für alle $x$ in $U^{(i)}$ $n_x^{(i)} \leq b_x|T^{(i)}|/|U^{(i)}|$ ist, wobei $b_x$ der Verzweigungsgrad des Knotens $x$ ist. Offensichtlich bedingt dies, daß sich die Verteilung der Knoten in $U^{(i)}$ der Verteilung der Knoten von $T^{(i)}$ anpaßt. Die Abbildung 3 zeigt einen Suchbaum mit $k$ gleichmäßig verteilten Kontrollknoten.

Dazu sind mehrere Strategien geeignet. Entweder nimmt man die Umstrukturierung von $U^{(i)}$ ggf. sofort nach der Expansion des $i$-ten Horizontknotens vor, oder erst dann, wenn nach mehreren Expansionen die maximale Kontrollgröße der Knoten einen geeignet gewählten Schwellwert überschreitet. In diesem Abschnitt wird zunächst der letztere Fall untersucht. SBIDA* folgt der letzten Strategie: eine Umstrukturierung, d. h. die Anpassung von $U^{(i)}$ an $T^{(i)}$, wird erst nach mehreren Expansionen durch einen einzigen Durchlauf durch den gesamten Baum vorgenommen.

Ähnlich wie Algorithmus MEIDA* arbeitet SBIDA* in zwei Phasen. In

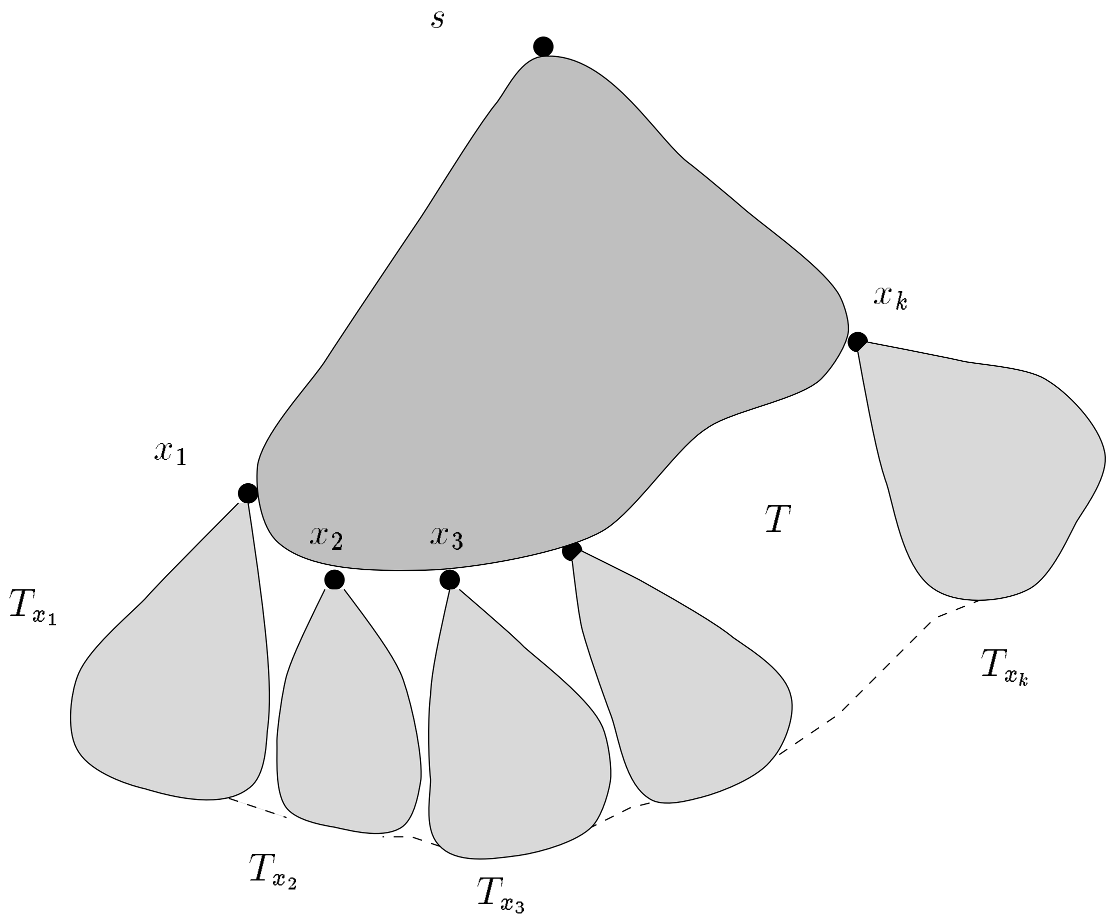


Abbildung 3: Der Suchbaum $T$ mit $k$ Kontrollknoten.

der ersten Phase expandiert SBIDA* zunächst eine Reihe von Horizontknoten von $U^{(i)}$, sagen wir z. B. $m$ viele, die in einer Min-Vorwärtswarteschlange $D$ abgelegt sind, und in der zweiten Phase werden die Knoten von $U^{(i)}$ neu verteilt. Nehmen wir an, daß die Funktion BAL diese Neuverteilung der zu speichernden Knoten vornimmt und diese entsprechend den $f_x$-Werten in $D$ speichert. Um die Darstellung zu vereinfachen, nehmen wir an, daß immer ein Zielknoten existiert. Dann läßt sich SBIDA* folgendermaßen beschreiben.

**Algorithmus** SBIDA*;<br>
**begin**<br>
    $s :=$ Startkonfiguration;<br>
    {führe A* durch bis der vorhandene Speicherplatz belegt ist}<br>
    $D :=$ A*_*limit*$(m)$;<br>
    {Schleifeninitialisierung};<br>
    $t := \infty$;<br>
    **while** ein Zielknoten ist nicht erreicht **do**

        {Phase 1}<br>
        **while** die Anzahl der Knotenexpansionen ist $< m$ **do**<br>
            $x := \min_{f_x}(D)$;<br>
            **if** $x$ ist ein Zielknoten **then exit**;<br>
            $t := f_x$; {speichere letzten Schwellwert}<br>
(1)     $n_x :=$ IDA*_*size_of_tree*$(s, t)$;<br>
(2)         $f_x :=$ IDA*_*next_thresh*$(x, t)$;<br>
            *D.increase_key*$(x, f_x)$;<br>
        **endwhile**

        {Phase 2}<br>
(3)     $n :=$ IDA*_*size_of_tree*$(s, t)$;<br>
(4)     BAL$(s, t, n/m)$;<br>
    **endwhile**<br>
**end**

In (1) wird mittels eines IDA*-Schritt die Größe des von $x$ kontrollierten Teilbaumes bestimmt, und in (2) dessen $f_x$-Wert. Natürlich lassen sich beide Werte mit einem IDA*-Schritt bestimmen. Die Prozedur IDA*_*size_of_tree* kann natürlich auch die Anzahl der bisher erzeugten Knoten aktualisieren. Diejenigen Knoten, deren $f^*$-Wert gleich dem Schwellwert $f_x$ ist, werden durch diesen IDA*-Schritt erstmals expandiert. Daher können die neu erzeugten Knoten als solche erkannt werden.

Die im folgenden vorgestellten Baumbalancierungsstrategien $BAL$ erzeugen, wie bei IDA*, einen durch den Schwellwert $t$ beschränkten Suchbaum. Da in der Phase 1 möglicherweise nicht alle Knoten expandiert worden sind, deren $f^*$-Wert gleich $t$ ist, wird zunächst die Größe dieses Suchbaumes in Schritt (3) bestimmt, d. h. die Anzahl aller erzeugten Knoten (nicht zu verwechseln mit $n_s$ die Anzahl der von $s$ kontrollierten Knoten). Bei der später folgenden Analyse wollen wir Schritt (3) vernachlässigen, da er keinen Einfluß auf die asymptotische Laufzeit hat, sondern die Laufzeit höchstens um einen konstanten Faktor erhöht. In der Tat kann auf den Aufruf von IDA*_*size_of_tree* in (3) ganz verzichtet werden. Wir werden nämlich eine Variante zu den Baumbalancierungsalgorithmen BAL kennenlernen, die ohne die Übergabe von $n/m$ und damit ohne die Angabe der aktuellen Größe des zu unterteilenden Suchbaumes auskommt.

## 5.3 Baumbalancierungsstrategien

Im folgenden stellen wir zwei Algorithmen zur Aufteilung eines Baums in gleich große Teilbäume vor und vergleichen sie im Hinblick auf ihre Anwendung in SBIDA*.

### 5.3.1 Baumaufteilung durch Abzählen

Der erste Algorithmus BAL-Version 1 verfolgt folgende Strategie: Man zählt sukzessiv von den Blättern zur Wurzel ausgehend die Anzahl der von einem beliebigen Knoten $x$ in $U^{(i)}$ kontrollierten Knoten. Die Anzahl der von $x$ kontrollierten Knoten ist gleich der Summe der von den Söhnen kontrollierten Knoten plus eins (ein Knoten kontrolliert sich selbst). Ist diese Summe größer oder gleich dem Wert $n^{(i)}/m$ (wobei $n^{(i)} = |T^{(i)}|$), so wird $x$ zum Kontrollknoten und gespeichert. Die von einem Kontrollknoten kontrollierten Knoten werden in der Summenbildung darüberliegender Level nicht mehr berücksichtigt.

Das Verfahrens läßt sich rekursiv wie folgt beschreiben:

**procedure** BAL-Version 1 ($s$: Startknoten, $t$: letzter IDA*-Schwellwert,
$\alpha$: Grenzwert für die Kontrollgröße);
**begin**
  make_empty($D$); {leere $D$}
  $(n_s, f_s) := \mathit{generate\_control\_nodes}(s, t, \alpha)$;
  $D$.insert$(s, f_s)$;
**end**

**function** *generate_control_nodes*($x$: Knoten, $t$: Schwellwert,
    $\alpha$: Grenzwert der Kontrollgröße) : (integer, integer);

**begin**
  **if** $f(x) > t$ **then** {$x$ ist ein Horizontknoten}
    return$(1, f(x))$;
  **else**
    $n_x := 1$;
    $f_x := \infty$;
    $S(x)$ :=expand$(x)$;
    **foreach** $x' \in S(x)$ **do**
      $(n_{x'}, f_{x'}) := \mathit{generate\_control\_nodes}(x', t, \alpha)$;
      $n_x := n_x + n_{x'}$ ;
      $f_x := \min(\{f_x, f_{x'}\})$;
    **endfor**
    **if** $n_x \geq \alpha$ **then** {neuer Kontrollknoten wird gebildet}
      $D$.insert$(x, f_x)$; {$n_x$ wird ebenfalls gespeichert}
      return$(0, \infty)$;
    **else**
      return$(n_x, f_x)$;
    **endif**
  **endif**
**end**

Daß BAL-Version 1 tatsächlich die gewünschte Unterteilung des Suchbaumes liefert, zeigt die folgende Überlegung. Jeder Kontrollknoten kontrolliert höchstens $b_x n^{(i)}/m$ viele Knoten, da jeder Sohnknoten höchstens $n^{(i)}/m - 1$ viele Knoten kontrolliert (denn sonst wäre der Sohnknoten selbst ein Kontrollknoten). Damit ergibt sich insgesamt: Da jeder Knoten nur genau einmal gezählt wird und jeder gespeicherter Kontrollknoten mindestens $n^{(i)}/m$ viele Knoten kontrolliert, folgt sofort daß höchstens $m$ viele Knoten gespeichert werden.

Es ist möglich, die Anzahl der von $x$ kontrollierter Knoten unabhängig von dem Verzweigungsgrad $b_x$ zu beschränken. Hierzu unterteilt man den Teilbaum mit Wurzel $x$ in Bereiche, die in der Generierung aufeinanderfolgender Sohnknoten von $x$ gebildet werden. Man zählt die von aufeinander folgenden Söhnen kontrollierten Knoten solange zusammen, bis die Summe größer oder gleich $n^{(i)}/m$ ist und setzt dann einen Kontrollknoten. Dieser ist durch das Tripel $(x, y, z)$ bestimmt, wobei $x$ der Kontrollknoten und $y$ bzw. $z$ der erste bzw. letzte der in diesem Bereich enthaltenen Söhne ist. Der Suchbaum läßt sich so in Bereiche der Größe $< 2n^{(i)}/m$ unterteilen. Auf diese Weise wird garantiert, daß mindestens ein Speicherplatz der Größe $m/2$ ausgenutzt wird.

Da bei diesem Verfahren keine Vorgängerknoten gespeichert werden, liegt jedoch beim Finden eines Zielknotens dessen Lösungsweg nicht vollständig vor. Der Lösungsweg muß durch nochmaliges Starten eines oder mehrerer IDA*-Schritte rekonstruiert werden. Das verschlechtert zwar nicht die asymptotische Laufzeit, ist aber bei konkreten Anwendungen auch nicht wünschenswert. Das folgende Baumaufteilungsverfahren vermeidet diesen Nachteil.

### 5.3.2 Baumaufteilung mit Vorgängerknoten

Der Nachteil von BAL-Version 1 besteht darin, daß in den Schritten (1) und (2) von SBIDA* vor der Expansion jedes Knotens $x$ geprüft werden muß, ob $x$ ein Kontrollknoten ist, d.h. ob $x$ in $D$ enthalten ist. Wenn das der Fall ist, wird die weitere Expansion an dieser Stelle abgebrochen. Unter Verwendung eines Hash-Verfahrens läßt sich die Suchoperation in

$D$, in $O(1)$ erwarteter Zeit ausführen. Dennoch kann dies in praktischen Anwendungen die Effizienz des Verfahrens merklich beeinträchtigen.

Um diesen Test zu vermeiden, speichern wir in BAL auch alle Vorgängerknoten von Kontrollknoten in dem Heap $D$. Ferner werden für jeden in $D$ eingefügten Knoten Zeiger auf den Vaterknoten, einen Bruderknoten und die Liste der Sohnknoten abgespeichert. Wir betrachten nun jeden Knoten in $D$ als Kontrollknoten. Damit können für jeden Knoten $x$ in $U^{(i)}$ effizient die Söhne, die keine Kontrollknoten sind, bestimmt werden und nur diese werden weiter expandiert, d.h. die weitere Expansion in den Schritten (1) und (2) von SBIDA* mit Startknoten $x$ wird dann und nur dann bei Sohn $x'$ abgebrochen, wenn $x'$ bereits in $D$ enthalten ist.

Sie $d$ die maximale Suchtiefe einer Probleminstanz und $n$ die Größe des zu erzeugenden Suchbaumes, der unterteilt werden soll. Knoten $x$, die die Ungleichung

$$\frac{d \cdot n}{m} \leq n_x \leq \frac{d \cdot b_x n}{m}$$

erfüllen, bezeichnen wir im folgenden als *Kontrollknoten im engeren Sinne* oder *$M$-Kontrollknoten* (engl. *main*), im Gegensatz zu den anderen Kontrollknoten, die Vorgänger von $M$-Kontrollknoten sind und als solche innere Knoten des gespeicherten Suchbaumes sind. $M$-Kontrollknoten können ebenfalls innere Knoten sein und müssen nicht notwendig Blätter des expliziten Graphen sein. Die Anzahl der $M$-Kontrollknoten sei $l$. Aus der obigen Ungleichung folgt, daß $l \leq m/d$. Damit ist $l \cdot d \leq m$, d. h. die Anzahl aller $M$-Kontrollknoten sowie deren Vorgänger ist höchstens $m$ und die Speicherung der Kontrollknoten erfordert daher einen Speicherplatz der Größe $m$. Der ungünstigste Fall tritt dann ein, wenn der erzeugte Suchbaum zu einer linearen Liste degeneriert, d. h. $d = \Theta(n)$. Dann gibt es nur einen $M$-Kontrollknoten und die Gesamtzahl aller Kontrollknoten ist $l \cdot d = \Theta(n)$. Für ausgeglichene Bäume ist die Gesamtzahl aller Kontrollknoten, einschließlich der Vorgängerknoten, jedoch von der Größenordnung $\Theta(l)$.

Das Verfahren kann in Pseudo-Pascal wie folgt beschrieben werden.

**procedure** BAL-Version 2 ($s$: Startknoten, $t$: Schwellwert
des letzten IDA*-Schrittes, $\alpha$: Kontrollgröße);

**begin**
  make_empty($D$); {leere $D$}
  $(saved, n_s, f_s)$ := *generate_control_nodes*$(s, t, \alpha)$;
  $D$.insert$(s, f_s)$;
**end**

**function** *generate_control_nodes* ($x$ : Knoten, $t$ : Schwellwert,
    $\alpha$ : Kontrollgröße) : (boolean, integer, integer);

**begin**
  **if** $f(x) > t$ **then** {$x$ ist ein Horizontknoten}
    return$(false, 1, f(x))$;
  **else**
    $n_x$ := 1;
    $f_x$ := $\infty$;
(5)    $S(x)$ :=expand$(x)$;
    **foreach** $x' \in S(x)$ **do**
      $p(x') := x$;
      $(saved, n_{x'}, f_{x'})$ := *generate_control_nodes*$(x', t)$;
      **if** saved **then** {Sohn von $x$ ist bereits gespeichert}
        markiere $x$;
      $n_x := n_x + n_{x'}$ ;
      $f_x := \min\{f_x, f_{x'}\}$;
    **endfor**
    **if** $n_x \geq \alpha$ **or** $x$ ist markiert **then**
      {neuer Kontrollknoten wird gebildet}
      $D$.insert$(x, f_x)$;
      return(true,0,$\infty$);
    **else**
      return(false,$n_x$,$f_x$);
    **endif**
  **endif**
**end**

Die Zuweisung des Schwellwertes $\alpha$ in Schritt (4) von SBIDA* muß natürlich zu $\alpha = dn/m$ geändert werden. Bei dieser Darstellung setzen wir voraus, daß für einen Kontrollknoten $x$ die *expand*-Funktion im Schritt (5) die Menge der Sohnknoten von $x$ zurückliefert, die nicht als Sohnknoten im expliziten Graphen gespeichert sind (diese lassen sich auch ohne Suchoperation effizient bestimmen). Für alle anderen Knoten sei $S(x) = \Gamma(x)$, d. h. auf einen Test auf Duplikate wird verzichtet.

Nach der obigen Beschreibung von BAL-Version 2, werden die Knoten von $T^{(i)}$ durchlaufen und alle Knoten von $U^{(i)}$ gespeichert. Da sich $U^{(i)}$ jedoch bei einem Aufruf von BAL-Version 2 nur geringfügig ändert, ist es sinnvoll BAL-Version 2 so zu modifizieren, daß nur noch der Teil von $U^{(i)}$ durchlaufen wird, der sich gegenüber $U^{(i-1)}$ geändert hat. Dies kann wie folgt erreicht werden. Anstatt *generate_control_nodes* an der Wurzel $s$ rekursiv aufzurufen, sucht man zunächst das am weitesten links liegende Blatt von $U^{(i-1)}$ auf und startet *generate_control_nodes* an diesem Blatt. Natürlich muß die rekursive Formulierung von *generate_control_nodes* in eine iterative umgewandelt werden, die sich auf die Zeigerstruktur der Knoten in $D$ stützt. Damit wird erreicht, daß nur die Horizontknoten und die zu löschenden Knoten von $U^{(i-1)}$ sowie die neu hinzukommenden Knoten von $U^{(i)}$ betrachtet werden müssen.

Da BAL nach $m$ vielen Expansionen von Horizontknoten von $T^{(i)}$ in SBIDA* aufgerufen wird, ist zu erwarten, daß sich $U^{(i)}$ gegenüber $U^{(i-1)}$ nur wenig verändert, womit auch nur wenige Update-Operationen durchgeführt werden müssen. Das stellt einen großen Effizienzvorteil gegenüber Verfahren wie MA* dar, die häufig für die Expansion eines einzigen Horizontknotens viele Update Operationen an Datenstrukturen, wie z. B. das Einfügen und Entfernen in eine Vorwärtswarteschlange, vornehmen müssen. Der Speicherplatzbedarf von BAL-Version 2 und damit von SBIDA* beträgt, wie bereits oben erwähnt, im Worst-Case $O(l \cdot d)$.

Diese Variante der Baumaufteilung mit Speicherung der Vorgängerknoten hat einen weiteren Vorteil, denn wie im folgenden ausgeführt, kommt man beim Aufruf der BAL-Prozedur ohne die aktuelle Größe des generierten Suchbaumes aus.

### 5.3.3 Baumaufteilung ohne Übergabe der aktuellen Baumgröße

Die Prozedur BAL bestimmt in einem Durchlauf durch den gesamten Baum die Kontrollknoten. Dazu wird beim Aufruf von BAL die Größe des Suchbaumes $n$ (bzw. $\alpha = n/m$) als Parameter übergeben. Werden die Vorgänger von $M$-Kontrollknoten gespeichert, kann die Menge der $M$-Kontrollknoten auch ohne die Übergabe dieses Wertes korrekt bestimmt werden. Es genügt, beim Durchlaufen des zu erzeugenden Suchbaumes entsprechend einer bestimmten Durchlaufordnung, die Anzahl der aktuell erzeugten Knoten in $n$ mitzuzählen. Dies läßt sich mittels des folgenden Lemmas zeigen.

**Lemma 5.3** *Es sei $T$ ein Suchbaum der Tiefe $d$ und $T^{(i)}$, $1 \leq i \leq k$, eine Folge von Teilbäumen von $T$ mit*

$$T^{(0)} \subset T^{(1)} \subset \ldots \subset T^{(k)} = T.$$

*Ferner sei $T_x^{(i)}$ der in $T^{(i)}$ enthaltene Teilbaum mit Wurzel $x$ und $rk(x)$—der* Rang *von $x$—die Anzahl der Knoten $x'$ in $T^{(i)}$ mit $|T_{x'}^{(i)}| \leq |T_x^{(i)}|$. Ist $U^{(i)}$ die Menge der $m$ Knoten $x$ mit höchstem Rang, d.h. für einen Knoten in $U^{(i)}$ ist $rk(x) \geq |T^{(i)}| - m$, so gilt für $1 \leq i \leq k$:*

$$n_x^{(i)} \leq \frac{db_x|T^{(i)}|}{m}.$$

Bevor wir das Lemma beweisen, sei angemerkt, daß wir den Begriff der Kontrollgröße in Abschnitt 5.2 prozedural eingeführt haben. Natürlich läßt sich dieser Begriff auch induktiv über den Aufbau eines Baumes $T$ in Abhängigkeit zu einem Teilbaum $U$ definieren, durch $n_x = 1 + \sum_{\{x'|p(x')=x, x' \in T-U\}} n_{x'}$, $x \in T$. Interpretiert man U als den expliziten Graphen, so stimmen diese beiden Definitionen überein.

**Beweis:** Seien die Voraussetzungen des Lemmas erfüllt. Wir zeigen zunächst, daß falls ein Knoten $x$ nicht in $U^{(i)}$ enthalten ist, dann gilt $|T_x^{(i)}| < d|T^{(i)}|/m$. Wir führen einen Widerspruchsbeweis. Hierzu sei $x \notin U^{(i)}$ mit $|T_x^{(i)}| \geq d|T^{(i)}|/m$. Dann gilt für alle Knoten $x'$ mit $rk(x') \geq |T^{(i)}| - m$: $|T_{x'}^{(i)}| \geq |T_x^{(i)}|$. Fassen wir die Menge $U^{(i)}$ als Baum auf, so enthält $U^{(i)}$ mindestens $m/d$ viele Blätter $l_1, \ldots, l_{m/d}$, da

$|U^{(i)}| = m$ und die maximale Tiefe von $U^{(i)}$ kleiner gleich $d$ ist. Nun gilt $|T^{(i)}| > \sum_{i=1}^{m/d} |T_{l_i}^{(i)}|$, da dieser Baum auch innere Knoten besitzt und damit erhalten wir

$$|T^{(i)}| > (m/d) \cdot |T_x^{(i)}| \geq (m/d) \cdot (d|T^{(i)}|/m) = |T^{(i)}|,$$

womit wir den Widerspruch erhalten.

Sei nun $x$ ein Knoten in $U^{(i)}$ mit Söhnen $y_1, \ldots, y_{b_x}$. O.B.d.A. seien die Söhne $y_1, \ldots, y_k$, $k \leq b_x$, nicht in $U^{(i)}$ enthalten. Nach der Bemerkung vor dem Beweis und dem vorigen Absatz gilt:

$$\begin{aligned} n_x^{(i)} &= 1 + \sum_{i=1} k n_{y_i}^{(i)} \\ &= 1 + \sum_{i=1} k |T_{y_i}^{(i)}| \\ &\leq 1 + kd|T^{(i)}|/m - k \\ &\leq b_x d|T^{(i)}|/m, \end{aligned}$$

was zu zeigen war. □

Die im vorigen Abschnitt vorgestellte Baumaufteilungsprozedur BAL erzeugt, wie das IDA*-Verfahren, der Reihe nach von links nach rechts gehend den durch einen Schwellwert beschränkten Suchbaum. Für jeden Knoten $x \in T$ des aktuellen Suchpfades wird rekursiv die Größe des Teilbaumes $T_x$ bestimmt (der evtl. zu diesem Zeitpunkt noch nicht vollständig erzeugt worden ist). Gilt $|T_x| > db_x n/m$ $(**)$, wobei $n$ die Größe des zu erzeugenden Suchbaumes ist, so wird $x$ zum Kontrollknoten.

Wir ändern dieses Vorgehen in der folgenden Weise ab: $n$ gebe die Anzahl der bis zu dem aktuellen Zeitpunkt erzeugten Knoten an. Denken wir uns die Erzeugung von $T$ auf naheliegende Weise in folgende Teilschritte zerlegt: Ausgehend von einem Schritt $i-1$ erzeugen wir im $i$-ten Schritt den weiteren Teil des durch den Schwellwert $t$ beschränkten Suchbaumes, bis ein Knoten $x$ gefunden wird, der die obige Ungleichung $(**)$ erfüllt (jetzt ist $n$ anders definiert!). Dieser wird in den expliziten Graphen $U^{(i)}$ aufgenommen. Muß nun im Gegenzug gegebenfalls ein anderer Knoten aus dem expliziten Graphen entfernt werden, wird der Blattknoten $y$ mit minimalem $n_y$-Wert gewählt. Damit sind die Voraussetzungen des obigen Lemmas erfüllt, woraus folgt, daß niemals ein $M$-Kontrollknoten aus dem expliziten Graphen entfernt wird. Also liefert diese entsprechend modifizierte Version der Prozedur BAL die gewünschte Unterteilung des Baumes.

Wird diese BAL-Prozedur verwendet, so kann man auf den Funktionsaufruf IDA*_*size_of_tree* in Schritt (3) verzichten. Erforderlich ist jedoch, daß durch die Prozedur *generate_control_nodes* eine weitere Vorwärtswarteschlange aktualisiert wird, in der alle Blattknoten des expliziten Graphen entsprechend ihrer $n_x$-Werten eingefügt werden. Wir gehen hier nicht näher auf diese Variante ein, da wir unter den dynamischen Baumbalancierungsalgorithmen, die wir in Abschnitt 5.4 noch genauer besprechen, eine ähnliche Technik anwenden.

Wir wollen daraufhin weisen, daß die obige Formulierung des Lemmas das Auftreten von Duplikaten nicht ausschließt. Um die Analysen zu vereinfachen, haben wir generell vorausgesetzt, daß keine Duplikate auftreten.

### 5.3.4 Zulässigkeit, Komplexität

In diesem Abschnitt weisen wir nach, daß SBIDA* zulässig ist. Anschließend bestimmen wir die Komplexität dieses Algorithmus.

Bisher war die Kontrollgröße $n_x$ als die Größe des in $T$ enthaltenen Teilbaumes mit Wurzel $x$ definiert worden. Im Hinblick auf eine Beschränkung der Komplexität, d. h. der Anzahl der Knotenexpansionen, ist es jedoch sinnvoll, $n_x$ als die Anzahl der inneren (=expandierten) Knoten des von $x$ kontrollierten Suchbaumes zu definieren. Die Algorithmen lassen sich leicht entsprechend dieser Definition von $n_x$ modifizieren. Wir führen die folgende Analyse unter dieser Voraussetzung durch. Da, unter der Annahme eines maximalen Verzweigungsgrades, die Anzahl der expandierten Knoten von der gleichen Größenordnung ist wie die Anzahl der erzeugten Knoten, ändert sich durch diese Modifikation nichts an der asymptotischen Komplexität. Die Analyse wird jedoch dadurch einfacher.

**Satz 5.4** *Der Algorithmus* SBIDA* *ist zulässig.*

**Beweis:** Wir wollen zeigen, daß SBIDA* geordnet ist, woraus mit Satz 2.2 sofort dessen Zulässigkeit folgt.

Zunächst zeigen wir, daß die in $D$ gespeicherten Horizontknoten von $T$ entsprechend ihrer $f^*$-Ordnung expandiert werden. Dies gilt, da $D$

folgende Invarianten erfüllt:

1. Ein Kontrollknoten $x \in D$ kontrolliert mindestens einen Horizontknoten von $y \in T$ mit $f^*(y) = f_x$, der noch nicht expandiert wurde. Alle Knoten $y \in T$ mit $f^*(y) < f_x$ wurden bereits expandiert.

2. Wird ein IDA*-Schritt von $x$ aus durchgeführt, so werden alle Knoten $y \in T_x$ mit $f^*(y) = f_x$ expandiert, d. h. insbesondere mindestens ein weiterer Knoten aus der *OPEN*-Liste.

3. Ein IDA*-Schritt wird immer von einem minimalen Element $x$ aus $D$ gestartet. Dieses wird entsprechend der neu erzeugten Horizontknoten in $D$ wieder aktualisiert.

Einerseits folgt aus (1.) und (2.), daß der Suchbaum mit jedem neuen IDA*-Schritt mindestens um einen Knoten erweitert wird, andererseits folgt aus (3.), daß die Horizontknoten entsprechend der ständig aktualisierten $f^*$-Ordnung expandiert werden. Also werden die Knoten von $T$ entsprechend ihrer $f^*$-Ordnung expandiert (evtl. mit Ausnahme von Duplikaten). Daraus folgt, daß die obigen Algorithmen geordnet sind und mit Satz 2.2 ergibt sich deren Zulässigkeit. $\square$

Es folgt eine Aussage über die Komplexität, gemessen in der Anzahl expandierter Knoten.

**Satz 5.5** *Der Algorithmus* SBIDA* *expandiert höchstens* $O(N_{\mathrm{A}^*}^2/m + N_{\mathrm{A}^*}m)$ *bzw.* $O(d \cdot N_{\mathrm{A}^*}^2/m + N_{\mathrm{A}^*}m)$ *viele Knoten unter Verwendung der Baumaufteilungsalgorithmen* BAL *bzw.* BAL*2.*

**Beweis:** Sei $I = (G, h)$ eine Probleminstanz aus $I_{CON}$ und $T$ der von A* und SBIDA* erzeugte Teilbaum (d. h. A* und SBIDA* verwenden dieselbe *tie-breaking*-Regel). Weiter sei $V$ die Menge der inneren Knoten von $T$ und $n := |V|$. Dann ist $n = N_{\mathrm{A}^*}(I)$.

a) Wir zeigen zunächst die Behauptung für SBIDA* mit BAL. Die Generierung von $T$ wird in folgende Teilschritte zerlegt:

Schritt:

$(1,0)$: Es werden $m$ viele innere Knoten des ereugten Graphen in $D$ entsprechend ihrem $f^*$-Wert gespeichert (ähnlich dem A*-Verfahren).

$(1,k)$, $0 < k \leq j_1$: Es wird ein IDA*-Schritt von dem Element $x$ in $D$ mit minimalem $f_x$-Wert gestartet, um den nächsten Horizontknoten zu expandieren. In den Schritten $(1,1),\ldots,(1,j_1)$ werden mindestens $m$ viele Horizontknoten expandiert.

$(2,0)$: Zunächst wird mittels einem IDA*_*size_of_tree*-Aufruf die Anzahl der inneren Knoten $n_2$ des Suchbaumes bestimmt, der durch den im Schritt $(1,j_{i-1})$ verwendeten Schwellwert beschränkt ist. In diesem Suchbaum werden durch einen globalen Verteilungsschritt der Prozedur BAL die maximal $m$ vielen $M$-Kontrollknoten bestimmt und in $D$ gespeichert. Nach dem Verteilungsschritt gilt für jeden $M$-Kontrollknoten $x$ die Kontrollgrößenbedingung, d. h.

$$n_2/m \leq n_x < b_x n_2/m \qquad (1).$$

$(2,k)$, $0 < k \leq j_2$: Verfahre wie in Schritt $(1,k)$.

...

$(i,0)$, $2 < i \leq l$: Verfahre wie in Schritt $(2,0)$, d. h. bestimme zunächst die Anzahl $n_i$ der inneren Knoten des Suchbaumes bestimmt, der durch den im Schritt $(i-1,j_{i-1})$ verwendeten Schwellwert beschränkt ist. Anschließend werden durch die Prozedur BAL die *ies*-Kontrollknoten gesetzt. Nach dem Verteilungsschritt gilt auch hier für jeden $M$-Kontrollknoten $x$ wieder die Kontrollgrößenbedingung:

$$n_i/m \leq n_x < b_x \cdot n_i/m \qquad (2).$$

$(i,k)$ , $1 \leq k \leq j_i$: Verfahre wie im Schritt $(1,k)$.

Es sei $T_j^{(i)}$ der nach Schritt $(i,j)$ erzeugte Graph und $V_j^{(i)}$ die Menge der inneren Knoten von $T_j^{(i)}$. Es sei $(l,j_l)$ der letzte Expansionsschritt, und $b$ der maximale Verzweigungsgrad. Dann erhalten wir die (bzgl. Inklusion) aufsteigend geordnete Folge von Graphen:

$$V_0^{(1)}, \ldots, V_{j_1}^{(1)}, V_0^{(2)}, \ldots, V_{j_2}^{(2)}, \ldots, V_0^{(l)}, \ldots, V_{j_l}^{(l)} = V.$$

Wir definieren ferner für $1 \leq i \leq l$ und $0 < k \leq j_i$: $S_k^{(i)} = V_k^{(i)} - V_{k-1}^{(i)}$.
Es gelten die folgenden Randbedingungen: Für $1 \leq i \leq l$,und $0 < k \leq j_i$ ist $|S_k^{(i)}| \geq 1$. Für $1 \leq i \leq l$ ist $|V_{j_i-1}^{(i)} - V_0^{(i)}| < m$ und $|V_{j_i}^{(i)} - V_0^{(i)}| \geq m$. Ferner gilt $|V_0^{(i)}| \geq |V_{j_{i-1}}^{(i-1)}|$.
Es folgt insbesondere, daß $|V_0^{(i)}| \geq i \cdot m$ und $l \leq \frac{n}{m}$. Die Anzahl der in Schritt $(i, 0)$ expandierten Knoten ist $2|V_0^{(i)}|$ und in Schritt $(i, k), k \neq 0$ durch $\frac{b}{m}|V_0^{(i)}| + |V_k^{(i)} - V_0^{(i)}|$ nach oben beschränkt. Damit ist

$$\begin{aligned}
N_{\mathrm{SBIDA}^*}(I) &\leq \sum_{i=1}^{l} (2|V_0^{(i)}| + \sum_{k=1}^{j_i} (b|V_0^{(i)}|/m + |V_k^{(i)} - V_0^{(i)}|)) \\
&= \sum_{i=1}^{l} ((2 + b/m \cdot j_i) \cdot |V_0^{(i)}| + \sum_{k=1}^{j_i} |T_k^{(i)} - V_0^{(i)}|) \\
&= \sum_{i=1}^{l} ((2 + b/m \cdot j_i) \cdot |V_0^{(i)}| + \sum_{k=1}^{j_i} \sum_{r=1}^{k} |S_r^{(i)}|) \\
&= \sum_{i=1}^{l} ((2 + b/m \cdot j_i) \cdot |V_0^{(i)}| + \sum_{k=1}^{j_i} (j_i + 1 - k) \cdot |S_k^{(i)}|)
\end{aligned}$$

$n_i := |T_0^{(i)}|$, $s_{i,k} := |S_k^{(i)}|$, $n_{l+1} := |V_{j_l}^{(l)}| = |V|$, so stellt folgende Funktion

$$\sum_{i=1}^{l} ((2 + \frac{b}{m} \cdot j_i) \cdot n_i + \sum_{k=1}^{j_i} (j_i + 1 - k) \cdot s_{i,k}) \qquad (*)$$

eine obere Schranke für die Komplexität dar. Wegen $s_{i,j_i} \leq n_{i+1} - n_i$ läßt sich der $i$-te Summand der oberen Summe wie folgt abschätzen:

$$\begin{aligned}
&(2 + b/m \cdot j_i) \cdot n_i + \sum_{k=1}^{j_i} (j_i + 1 - k) \cdot s_{i,k} \\
&\leq (2 + b/m \cdot j_i) \cdot n_i + (n_{i+1} - n_i) + \sum_{k=1}^{j_i-1} (j_i + 1 - k) \cdot s_{i,k} \\
&\leq (1 + b/m) \cdot j_i \cdot n_i + n_{i+1} + \sum_{k=1}^{j_i-1} (j_i + 1 - k) \cdot s_{i,k}
\end{aligned}$$

$$
\begin{aligned}
&\leq \ (1+b)\cdot n_i + n_{i+1} + \sum_{k=1}^{j_i-1} (j_i+1-k)\cdot s_{i,k} \\
&\leq \ (1+b)\cdot n_i + n_{i+1} + \sum_{k=1}^{j_i-1} (j_i+1-k)\cdot s_{i,k}.
\end{aligned}
$$

Damit erhalten wir

$$
N_{\text{SBIDA}^*}(I) \leq \sum_{i=1}^{l} n_{i+1} + (1+b)\cdot \sum_{i=1}^{l} n_i + \sum_{i=1}^{l}\sum_{k=1}^{j_i-1} (j_i+1-k)\cdot s_{i,k}
$$

Ähnlich wie im Beweis von Satz 5.2 lassen sich die folgenden Summen nach oben abschätzen durch

$$
\begin{aligned}
\sum_{i=1}^{l} n_{i+1} &\leq \sum_{i=1}^{n/m} (i+1)m = (n/m+2)(n/m+1)m/2 \\
(1+b)\cdot \sum_{i=1}^{l} n_i &\leq (1+b)\cdot \sum_{i=1}^{n/m} im = (1+b)(n/m+1)n/m\cdot m/2 \\
\sum_{i=1}^{l}\sum_{k=1}^{j_i-1}(j_i+1-k)\cdot s_{i,k} &\leq \sum_{i=1}^{n/m}\sum_{k=1}^{m-1}(m+1-k) \\
&= \sum_{i=1}^{n/m}(m(m-1)/2-1) \\
&\leq n(m-1)/2
\end{aligned}
$$

Damit erhalten wir für die Anzahl der Knotenexpansionen die folgende obere Abschätzung:

$$
\begin{aligned}
N_{\text{SBIDA}^*}(I) &\leq \frac{n^2}{2m} + \frac{3n}{2m} + m + (1+b)(\frac{n^2}{2m} + \frac{n}{2m}) + \frac{1}{2}n(m-1) \\
&= \frac{1}{2}(2+b)\frac{n^2}{m} + \frac{1}{2}n(m-1) + \frac{1}{2}(4+b)n + m \in O(\frac{n^2}{m} + nm).
\end{aligned}
$$

b) Wird SBIDA* zusammen mit BAL2 angewendet, d. h. mit Speicherung der Vorgängerknoten, so erfüllen nach einem Aufteilungsschritt die $M$-Kontrollknoten die etwas schwächere Bedingung

$$
dn_i/m \leq n_x < db_x n_i/m, \qquad (2)
$$

wobei $d$ die Lösungstiefe ist. Das bedeutet insbesondere, daß jeder Kontrollknoten $x$, also auch ein Vorgänger eines $M$-Kontrollknotens, höchstens $db_x n_i/m$ viele Knoten kontrolliert.

Die obere Herleitung ändert sich nur geringfügig. Wie leicht zu sehen ist, ergibt sich für die Komplexität von SBIDA* wie behauptet $N_{\mathrm{SBIDA}^*}(I) \in O(\frac{dn^2}{m} + nm)$. □

In unserem Algorithmus sind wir davon ausgegangen, daß erst mindestens $m$ viele Horizontknoten expandiert werden, bevor der Suchbaum neu unterteilt wird. Ist die Speicherplatzgröße $m$ im Vergleich zur Problemgröße sehr groß, d. h. ist $m > \sqrt{n}$, ist es ratsam die Minimalanzahl zu expandierender Horizontknoten zwischen zwei Aufteilungsschritten zu verringern. Führt man die Analyse analog zu oben für eine Minimalzahl $r \leq m$ durch, so erhält man die Komplexität $N_{\mathrm{SBIDA}^*}(I) = \frac{n^2}{2r} + \frac{1}{2}rn + \frac{br}{2m} + O(n)$, wobei $n = N_{\mathrm{A}^*}$ ist. Die reelle Funktion $F(r) = \frac{n^2}{2r} + \frac{1}{2}rn + \frac{br}{2m}$ nimmt für die positiven reellen Werte das Minimum bei $r = \frac{n}{\sqrt{b/m+n}} \approx \sqrt{n}$ an. Das ergibt die Komplexität $O(n \cdot \sqrt{n})$ für $m > \sqrt{n}$. Kennt man die Größenordnung der Problemgröße $n$ nicht, so kann man $r$ sukzessiv von Schritt zu Schritt auf $m$ steigern bis schließlich in einem Schritt $i$ mit $\sqrt{n_i} \approx m$ $r = m$ gewählt wird.

## 5.4 Dynamische Baumbalancierung

### 5.4.1 DBIDA*-Version 1

Die Idee von Algorithmus DBIDA* (Dynamic Balanced IDA*) besteht darin, daß man im Gegensatz zu SBIDA* die Umstrukturierung des expliziten Graphen $U^{(i)}$, falls notwendig, dynamisch an den erzeugten Suchbaum $T^{(i)}$ anpaßt. Dies wird durch folgende Strategie erzielt: Wie bisher wird der nächste Horizontknoten durch den Aufruf eines IDA*-Schrittes von einem der Kontrollknoten $x$ expandiert. Die Kontrollgröße $n_x$ wird dabei als Funktionswert zurückgeliefert. Ist der dabei erzeugte Teilbaum mit Wurzel $x$ zu groß, d. h. wird die Bedingung

$$n_x \leq \frac{db_x n}{m} \qquad (***)$$

verletzt, wobei $n$ die Anzahl der inneren (d. h. expandierten) Knoten des bisher erzeugten Suchbaumes sei, $b_x$ bzw. $d$ sei wie bisher der Verzweigungsgrad von $x$ bzw. die Lösungstiefe ist und $n_x$ die Anzahl der inneren Knoten des Teilbaumes mit Wurzel $x$, so werden diejenigen Sohnknoten von $x$ in $U^{(i)}$ aufgenommen, die mehr als $dn/m$ viele Knoten kontrollieren (mindestens ein solcher existiert). Diesen Vorgang nennen wir *Splitting.* Wir wollen darauf aufmerksam machen, daß hier die in Abschnitt 5.3.4 vorgeschlagene Definition für die Kontrollgröße $n_x$ verwenden. Wird ein Speicherplatz fester Größe verwendet und ist dieser bereits vollständig belegt, so werden im Gegenzug genauso viele Knoten aus $U^{(i)}$ wieder entfernt, wie eingefügt worden sind, und zwar werden die Knoten mit der geringsten Kontrollgröße entfernt, d. h. die Knoten $x$, für die $n_x$ minimal ist. Diesen zum Splitting inversen Vorgang nennen wir *Merging.*

Die dynamische Veränderung von $U^{(i)}$ läßt sich als Folge von Splitting und Merging Vorgängen beschreiben. Wie im vorangegangenem Abschnitt ausgeführt, wird niemals ein Knoten entfernt, der die Bedingung

$$n_x \geq \frac{db_x n}{m}$$

erfüllt.

Bei der folgenden Beschreibung von DBIDA* gehen wir davon aus, daß zu jedem Kontrollknoten $x$ auch ein Verweis auf dessen direkten Vorgänger $p(x)$ gespeichert wird, sowie der Wert $f_x$ und die Kontrollgröße $n_x$. Die Knoten von $U^{(i)}$ werden entsprechend ihrer $f_x$-Werte in einen Min-Heap $D$ und die Blätter von $U^{(i)}$ entsprechend ihrer $n_x$-Werte in einem Min-Heap $E$ gespeichtert. Ein Aufruf der Form $E.insert(x, n_x)$ bedeutet, daß der Knoten $x$ in $E$ entsprechend dem Wert $n_x$ eingefügt wird. $D$ enthält alle Knoten des expliziten Graphen, auch Knoten die keine Blätter von $T^{(i)}$ kontrollieren und deren $f^*$-Wert daher auf $\infty$ gesetzt ist. Wir nehmen der Einfachheit halber an, daß es mindestens einen Zielknoten gibt und keine Terminalknoten außer den Zielknoten auftreten.

**Algorithmus** DBIDA*;
**begin**
$s :=$ Startknoten;

$(D, E) :=$ A$^*$_$limit(m)$; {führe A$^*$ durch bis $m$ Knoten
gespeichert sind}
$n := m$; {$n$ zählt Anzahl der expandierten Knoten}
**while** $D$ ist nicht leer **and** ein Zielknoten ist nicht erreicht **do**
$x := \min_{f_x}(D)$;
$split(x, f_x)$;
**endwhile**
**end**;

Im folgenden beschreiben wir wie eine Split-Operation für Kontrollknoten durchgeführt werden kann. Hierzu wird zunächst ein IDA$^*$-Schritt von $x$ mit Schwellwert $f_x$ gestartet, um den nächsten Horizontknoten zu expandieren. Gleichzeitig werden dabei die Kontrollgröße und der $f_x$-Wert aller Söhne bestimmt sowie die Anzahl $n$ der expandierten Knoten aktualisiert. Ist der dabei erzeugte Teilbaum mit Wurzel $x$ zu groß, werden ein oder mehrere Söhne von $x$ in den expliziten Graphen eingefügt. Ist der Speicherplatz bereits vollständig belegt, so werden andere Blattknoten aus dem expliziten Graphen entfernt, d. h. sowohl aus $D$ als auch aus $E$. War $x$ selbst ein Blattknoten des expliziten Graphen, so muß $x$ aus $E$ entfernt werden, da mindestens ein Sohn von $x$ nun ein Blattknoten wird. Alle Söhne von $x$, deren Kontrollgröße $\geq dn/m$ ist, werden in $U^{(i)}$ eingefügt. Schließlich werden die Kontrollgrößen der nicht-eingefügten Söhne aufsummiert und bilden die Kontrollgröße des Knotens $x$, der ebenfalls in $U^{(i)}$ wieder eingefügt wird, da auch die Vorgänger aller Kontrollknoten in $U^{(i)}$ enthalten sind.

**procedure** split($x$: Knoten, $t$: integer) : integer;
**begin**
$S(x) := expand(x)$;
**foreach** $x' \in S(x)$ **do** {bestimme die Kontrollgröße
der Söhne von $x$}
(1) $n_{x'} :=$ IDA$^*$_$size_of_tree(x', t)$; {aktualisiert $n$}
$f_{x'} :=$ IDA$^*$_$next_thresh(x', t)$;
**endfor**;

```
        if Σ_{x'∈S(x)} n_{x'} > db_x n/m then
            {mindestens ein Sohn von x wird in D und E eingefügt}
            f_x := ∞;
            n_x := 1;
            if x ∈ E then E.delete(x);
            foreach x' ∈ S(x) do
(2)             if n_{x'} > dn/m then {füge diesen Knoten in D und E ein}
                    D.insert(x', f_{x'});
                    E.insert(x', n_{x'});
                    if |D| > m then {Speicherplatzmangel}
                        { entferne einen Knoten } merge;
                else { x' wird nicht in den expliziten Graphen eingefügt}
                    f_x := min{f_x, f_{x'}};
                    n_x := n_x + n_{x'};
                endif
            endfor;
        else if x ∈ E then {aktualisiere x in E}
            E.increase_key(x, n_x);
        endif
        D.increase_key(x, f_x);
    end;
```

Die folgende Prozedur *merge* entfernt den Knoten $x$ mit minimalem $n_x$-Wert aus dem expliziten Graphen, d. h. aus den Vorwärtswarteschlangen $D$ und $E$. Da dann der Vaterknoten $p(x)$ auch die Horizontknoten von $x$ kontrolliert, muß ggf. der Wert $f_{p(x)}$ heruntergesetzt und $n_{p(x)}$ um die Größe $n_x$ erhöht werden. Besitzt $p(x)$ keinen weiteren Sohnknoten mehr, so ist $p(x)$ wieder ein Blatt des expliziten Graphen und wird wieder in $E$ eingefügt.

```
    procedure merge; {entfernt ein Element aus den Datenstrukturen}
    begin
        x := E.access_min;
        f_{p(x)} := min{f_{p(x)}, f_x};
```

```
        n_p(x) := n_p(x) + n_x';
        {entferne x aus den verwendeten Datenstrukturen}
        D.delete(x);
        E.delete(x);
        {aktualisiere p(x) in den verwendeten Datenstrukturen}
        D.decrease_key(p(x), f_p(x));
        if letzter Sohn von p(x) wurde entfernt then
            E.insert(p(x), n_p(x));
    end
```

Wir setzen voraus, daß die Funktion IDA$^*$_*size_of_tree* die globale Variable $n$, die Anzahl der bisher insgesamt erzeugten Knoten, aktualisiert. Das läßt sich dadurch erzielen, daß beim Aufruf von IDA$^*$_*size_of_tree* $n$ um die Anzahl neu erzeugter Knoten erhöht wird; das sind genau diejenigen Knoten, deren $f^*$-Wert gleich dem Schwellwert $t$ ist. Wir nehmen an, daß die Funktionen IDA$^*$_*size_of_tree* oder IDA$^*$_*next_thresh* die weitere Expansion sofort abbrechen, wenn sie mit einem Startknoten aufgerufen werden, der bereits Kontrollknoten, d. h. Element von $D$, ist.

Der obige Algorithmus kann leicht für den Fall erweitert werden, daß beliebige Terminalknoten auftreten. Besteht der Horizont eines Teilbaumes mit Wurzel $x$ ausschließlich aus Terminalknoten, so liefert der IDA$^*$-Schritt IDA$^*$_*size_of_tree* den Schwellwert $\infty$ zurück. Ist $x$ ein Blatt des expliziten Graphen, so könnte $x$ aus $D$ entfernt werden, sobald wieder Speicherbedarf besteht. Auch hier gilt wieder das bei jedem Aufruf von IDA$^*$_*size_of_tree* oder IDA$^*$_*next_thresh* mindestens ein weiterer Horizontknoten expandiert wird; dies folgt, da immer ein Knoten $x$ gewählt wird, dessen $f_x$-Wert minimal ist. Ein Knoten, der einen Baum kontrolliert, dessen Blätter nur aus Terminalknoten bestehen, hat einen $f_x$-Wert von $\infty$ und wird deshalb nicht von DBIDA$^*$ zur Expansion ausgewählt.

Die oben beschriebene Version von DBIDA$^*$ ist konzeptuell einfach, und daher auch einfach zu implementieren. Ein Nachteil dieses Verfahrens ist, daß mit jedem neuen IDA$^*$-Schritt höchstens ein Splitting durchgeführt wird. Dadurch erfolgt die Anpassung an die Kontrollgrößenbedingung evtl. mit Verzögerung. Im folgenden Abschnitt wollen wir ein weiteres Verfahren vorstellen, daß die Einhaltung der obigen Kontrollgrößenbedingung $(***)$ dynamisch erzielt.

Eine Variante zu obigem Verfahren erhält man, wenn das Einfügen weiterer Knoten in den expliziten Graphen weniger restriktiv gehandhabt wird. Offenbar besteht hier ein Tradeoff: je weniger restriktiv das Einfügekriterium ist, um so eher müssen Knoten aus dem expliziten Graphen entfernt werden. So kann z. B. die Bedingung in Schritt (2) durch $n_x \geq b_x n/m$ ersetzt werden. Wie wir bereits gesehen haben, garantiert dies jedoch nicht die Einhaltung der Kontrollgrößenbedingung $n_x \leq b_x n/m$ für einen $M$-Kontrollknoten $x$. Bestenfalls läßt sich $n_x \leq d b_x |T|/m$ erreichen.

### 5.4.2 DBIDA*-Version 2

Da DBIDA* mit jedem IDA*-Schritt höchstens ein Splitting durchführt, kann die Kontrollgrößenbedingung $(***)$ aus Abschnitt 5.4.1 für jede weitere Expansion eines Horizontknotens nicht garantiert werden. Wir stellen nun ein dynamisches Verfahren vor, das diese Bedingung gewährleistet. Die Idee ist dabei, falls nötig, eine Kaskade von Splitting-Operationen durchzuführen. Jeder Aufruf eines IDA*-Schrittes führt zu weiteren rekursiven Aufrufen eines IDA*-Schrittes. Ist der von einem Knoten $x$ gestartete IDA*-Schritt beendet, so führt das in der Rekursionstiefe einen Schritt höher. Bevor das geschieht, wird nun geprüft, ob die Kontrollgrößenbedingung $(***)$ in Abschnitt 5.4.1 verletzt ist. Falls ja, wird der Knoten $x$ und mit ihm anschließend auch alle seine Vorgänger im expliziten Graphen gespeichert. Bei der folgenden Beschreibung des Verfahrens nehmen wir wieder an, daß es mindestens einen Zielknoten gibt und daß es keine Terminalknoten außer den Zielknoten gibt.

```
Algorithm DBIDA*2;
begin
    s := Startkonfiguration;
    (D, E) := A*_limit(m); {führe A* durch bis der vorhandene
                    Speicherplatz ausgenutzt ist}
    n := m;
    while ein Zielknoten ist noch nicht gefunden do
```

$x := \min_{f_x}(D)$;
**if** $x \in E$ { $x$ ist ein Blatt } **then** $E.delete(x)$;
$(f_x, n_{x'}) :=$ IDA$^*$_$bal(x, f_x)$;
**endwhile**
**end**

**function** IDA$^*$_bal($x$ : Knoten, $t$ : integer {Schwellwert} ):
(boolean, integer, integer);
**begin**
**if** $f(x) > t$ **then** { $x$ ist ein Horizontknoten }
return(false,0,$f(x)$);
**else**
**if** $f^*(x) = t$ **then** { $x$ wird erstmals expandiert }
(1) $n := n + 1$;
$n_x := 0$;
$f_x := \infty$;
$S(x)$ :=expand($x$);
**foreach** $x' \in S(x)$ **do**
$(saved, f_{x'}, n_{x'})$ :=IDA$^*$_bal($x'$,$f_s$);
$n_x := n_x + n_{x'}$ ;
$f_x := \min(\{f_x, f_{x'}\})$;
**if** $saved$ **then** { $x$ ist Vorgänger eines Kontrollknotens }
markiere $x$;
**endfor**
**if** $x \in D$ **then** {$x$ ist bereits gespeichert}
$D.increase\_key(x, f_x)$
**else**
**if** $x$ ist markiert **then**
{$x$ ist Vorgänger eines $l$-Kontrollknotens }
$D.insert(x, f_x)$;
**if** Speicherplatzüberlauf **then**
{ entferne einen Knoten } $merge$;
return(true,$\infty$, 0);
(2) **else**

**if** $n_x \geq dn/m$ **then**
{ $x$ ist Blatt und erfüllt die Kontrollgrößenbedingung}
(3) $D.insert(x, f_x)$;
(3′) $E.insert(x, n_x)$;
**if** Speicherplatzüberlauf **then**
{ entferne einen Knoten } *merge*;
(4) return(true,$\infty$, 0);
**else** return(false,$f_x$, $n_x$);
**endif**
**endif**
**endif**
**endif**
**end**

**procedure** *merge*; {entfernt ein Element aus den Datenstrukturen}
**begin**
(5) $x := E.access\_min$;
$f_{p(x)} := \min\{f_{p(x)}, f_x\}$;
$n_{p(x)} := n_{p(x)} + n_{x'}$;
{entferne $x$ aus den verwendeten Datenstrukturen}
$D.delete(x)$;
$E.delete(x)$;
{aktualisiere $p(x)$ in den verwendeten Datenstrukturen}
$D.decrease\_key(p(x), f_{p(x)})$;
**if** letzter Sohn von $p(x)$ wurde entfernt **then**
$E.insert(p(x), n_{p(x)})$;
**end**

Die globale Variable $n$, die die Anzahl der Knoten des bereits erzeugten Baumes ist, wird durch Schritt (1) in IDA*_balance aktualisiert. Der Algorithmus läßt sich auch leicht auf den Fall übertragen, daß auch Terminalknoten, die nicht Zielknoten sind, auftreten. Es sei noch daraufhin gewiesen, daß der in den Schritten (3) und (3′) eingefügte Blattknoten kein bzgl. der Kontrollgröße minimales Element in $E$ ist und daher nicht in der evtl. anschließenden *merge*-Prozedur wieder aus dem expliziten Graphen enfernt wird. Bei einem weniger restriktiven Einfügekriterium in (2) kann das durchaus der Fall sein. Dann muß der Return-Wert in (4) natürlich auf false gesetzt werden.

Wir haben die Kontrollgrößenbedingung in Schritt (2) zum entscheidenden Kriterium dafür gemacht, ob ein Knoten in (3) und (3′) bzw. in (5) gespeichert bzw. entfernt wird. Natürlich sind hier auch andere Kriterien denkbar: Wird ein Knoten $x$ gespeichert, wenn dessen $f_x$-Wert kleiner als der maximale $f$-Wert ($< \infty$) der gespeicherten Kontrollknoten ist, und in (5) der Blattknoten $y$ des expliziten Graphen mit maximalem $f_y$-Wert entfernt wird, so erhält man einen Algorithmus, der eine ähnliche Verteilung der gespeicherten Knoten vornimmt wie SMA*. Dieses Verfahren hat gegenüber SMA* jedoch den entscheidenden Vorteil, daß nur wenige Update-Operationen in der *OPEN* und *CLOSED*-Liste vorgenommen werden müssen und mehr Freiheit in der Wahl des nächsten zu expandierenden Horizontknotens besteht. Offenbar werden hier nicht nur ausschließlich Nachfolger von Blättern des expliziten Graphen expandiert. Es ist auch klar, daß dieses Verfahren zulässig ist. Wir wollen diese möglichen Variante hier nicht näher untersuchen, sondern auf eine nachfolgende Arbeit verweisen.

### 5.4.3 Zulässigkeit, Komplexität

Zunächst zeigen wir wieder die Zulässigkeit der Algorithmen DBIDA* und DBIDA*2, und anschließend wird die Komplexität von DBIDA*2 bestimmt.

**Satz 5.6** *Die Algorithmen* DBIDA* *und* DBIDA*2 *sind zulässig.*

**Beweis:** Ähnlich wie in vorangegangenen Beweisen läßt sich auch hier leicht einsehen, daß die Knoten von $T$ in $f^*$-Ordnung expandiert werden, woraus wieder mit Satz 2.2 sofort die Zulässigkeit dieser Verfahren folgt. □

Im folgenden machen wir eine Aussage zur Komplexität von DBIDA*2. Im Gegensatz zu SBIDA* ist für jeden IDA*-Schritt zur Expansion eines weiteren *OPEN*-Knotens gewährleistet, daß höchstens $db_x n/m$ viele bereits expandierte Knoten nochmals expandiert werden. Das folgt mit

Lemma 5.2. Daher ist eine geringere Anzahl expandierter Knoten zu erwarten.

**Satz 5.7** *Der Algorithmus* DBIDA*2 *expandiert* $O(d \cdot N_{\mathrm{A}^*}^2/m)$ *viele Knoten.*

**Beweis:** Der Beweis geht ähnlich wie die vorangegangenen Komplexitätsbeweise. Sei $I = (G, h)$ eine Probleminstanz aus $I_{CON}$ und $T$ der sowohl von DBIDA*2 als auch A* erzeugte Suchbaum. Es sei $V$ die Menge der inneren Knoten von $T$ und $n := |V|$. Dann ist also $n = N_{\mathrm{A}^*}$.

DBIDA*2 generiert im Schritt 0 den Graph soweit, bis $m$ Kontrollknoten gespeichert sind. Anschließend wird in jedem weiteren Schritt $i$ ein IDA*-Schritt von einem der Kontrollknoten gestartet, um mindestens einen weiteren Horizontknoten zu expandieren. Es sei $T^{(i)}$ der generierte Graph nach dem $i$-ten Schritt, $V^{(i)}$ die Menge der inneren Knoten von $T^{(i)}$. Wir erhalten wiederum eine Folge von (bzgl. Inklusion) aufsteigend sortierten Suchbäumen ($l$ sei wieder der letzte Schritt):

$$V^{(0)}, V^{(1)}, \ldots, V^{(i)}, \ldots, V^{(l)} = V.$$

Es sei $S^{(i)} := V^{(i)} - V^{(i-1)}, i > 0$. Im Gegensatz zu dem Beweis von Satz 5.5 läßt sich hier mittels der Anwendung von Lemma 5.3 die Kontrollgröße schärfer nach oben abschätzen: Nach jedem Schritt $i$ gilt: $n_x^{(i)} \leq db_x|V^{(i)}|/m$ für alle Kontrollknoten $x$, wobei $n_x^{(i)}$ die Anzahl der Knoten des Teilbaumes mit Wurzel $x$ in $V^{(i)}$ ist. Ist $b$ wieder der maximale Verzweigungsgrad, so werden in einem Schritt $i$ höchstens $d \cdot b\frac{|V^{(i-1)}|}{m} + |S^{(i)}|$ viele Knoten expandiert. Es folgt:

$$\begin{array}{rcl} N_{\mathrm{DBIDA^*2}}(I) & \leq & |V_0| + \Sigma_{i=1}^{l}\,(\frac{db|V^{(i-1)}|}{m} + |S^{(i)}|) \\ & = & |V^{(0)}| + \Sigma_{i=1}^{l}\ |S^{(i)}| + \Sigma_{i=1}^{l}\ \frac{db|V^{(i-1)}|}{m} \\ & = & |V| + \frac{db}{m} \cdot \Sigma_{i=1}^{l}\ |V^{(i-1)}| \end{array}$$

Ähnlich wie im Beweis zu Satz 5.2 ergibt sich auch hier, daß

$$\Sigma_{i=1}^{l}\ |V^{(i-1)}| \leq \Sigma_{i=m+1}^{|V|}\, i < \tfrac{1}{2}|V|(|V|+1).$$

Damit erhalten wir $N_{\mathrm{DBIDA^*2}}(I) \in O(dn^2/m)$.

□

## 5.5 Erweiterung zu Graph-Suchalgorithmen

Wir haben der Einfachheit halber unsere Algorithmen zunächst als Baum-Suchalgorithmen vorgestellt. Sie lassen sich aber sehr leicht zu Graph-Suchalgorithmen erweitern. Echte Graph-Suchalgorithmen benötigen eine Strategie beim Auftreten von neu erzeugten Duplikaten im Suchbaum. Natürlich arbeiten die Baum-Suchalgorithmen auch auf Graphen korrekt, also dann, wenn Duplikate nicht erkannt und gesondert behandelt werden. In diesem Fall nutzen sie die vorhandene Möglichkeit zur Beschneidung des Suchbaumes nicht aus. Prinzipiell bieten sich nur die Verfahren für eine Erweiterung an, die zu jedem Knoten auch alle seine Vorgänger speichern, wie z. B. DBIDA* und DBIDA*-2.

Folgende Strategie erweitert einen Baum-Suchalgorithmus zu einem Graph-Suchalgorithmus, ohne daß dessen Zulässigkeit verletzt wird. Bevor ein neu erzeugter Knoten $z$ in den expliziten Graphen eingefügt wird, wird geprüft, ob $z$ ein Duplikat ist. Angenommen das ist der Fall, d. h. es gibt einen Knoten $y$, der die gleiche Spielstellung wie $z$ repräsentiert. Ist $z$ auf einem teuereren Weg erzeugt worden, so wird $z$ wieder aus dem Suchbaum entfernt. Ist $y$ jedoch teurer, so wird dieser Knoten entfernt. Ist $y$ ein Blatt im expliziten Graphen, so kann das, ähnlich wie in der *merge*-Prozedur, geschehen: der Knoten wird entfernt und $p(y)$ wird durch Erhöhen des $n_y$-Wertes und ggf. des $f_y$-Wertes aktualisiert. Ist der Knoten jedoch kein Blattknoten, so muß mit $y$ auch der gesamte Teilbaum mit Wurzel $y$ entfernt werden. Das läßt sich durch einen IDA*-Schritt mit Startknoten $y$ realisieren, der diesen Teilbaum durchläuft und dabei alle Knoten von unten nach oben, ähnlich wie in der *merge*-Prozedur, aus dem expliziten Graphen entfernt. Die $n_x$- bzw. $f_x$-Werte werden dabei aktualisiert. Schließlich endet der IDA*-Schritt an der Wurzel $y$ und auch dieser Knoten kann nun entfernt werden. Führt man in einem IDA*-Schritt generell den Test auf Duplikate durch, so läßt sich das erneute Expandieren des entfernten Teilbaumes mit Wurzel $y$ vermeiden, solange $z$ im expliziten Graph bleibt. Wird dieser Test jedoch in einem IDA*-Schritt nicht durchgeführt, so läßt sich auch das Durchlaufen bereits erzeugter Teilbaume nicht vermeiden, wie in unserem Beispiel den Teilbaum mit Wurzel $y$. Im Hinblick auf die Anzahl expandierter Knoten, kann es daher sinnvoll sein auch Duplikate, deren Kontrollgrößenwert hoch ist, abzuspeichern. Um das Starten eines IDA*-Schrittes von einem teuren Duplikat, wie im oben genannten Beispiel der Knoten $y$, zu vermeiden, erhält dieser Wert in der Vorwärtswarteschlange den Wert $\infty$ zugewiesen.

# 6 Schluß

Wir haben in diesem Bericht die heuristischen Suchverfahren MEIDA*, DBIDA* und SBIDA* vorgestellt, die unter beschränktem Speicherplatz arbeiten. Sie nutzen einen Speicherplatz der Größe $m$ zur Reduzierung der Anzahl der Knotenexpansionen auf $O(n^2/m)$ bzw. $O(dn^2/m)$, wenn $n$ die Anzahl der von A* expandierten Knoten und $d$ die Suchtiefe ist. Alle anderen bisher in der Literatur veröffentlichten Algorithmen expandieren im schlimmsten Fall bis zu $\Omega(n^2)$.

Die Algorithmen IDA*, IDA*_CR, RBFS u. a. benötigen nur einen zur Lösungstiefe linearen Speicherplatz, aber können weitere Speicherplatzresourcen nicht zu einer Reduzierung der Komplexität ausnutzen. Algorithmus DBIDA* läßt sich problemlos mit anderen Ansätzen kombinieren, was zu einer weiteren Reduzierung der Knotenexpansionen führen kann.

Der Algorithmus DBIDA* weist zwei entscheidende Vorteile gegenüber Algorithmen wie MA* u. a. auf, die zwar auch eine vollständige Ausnutzung der Speicherplatzresourcen anstreben, aber bevorzugt die Knoten in unmittelbarer Nähe des zuletzt expandierten Horizontknotens in den Speicher einfügen. Der erste besteht darin, daß nur wenige Update-Operationen an den verwendeten Datenstrukturen, wie das Einfügen oder Entfernen von Knoten in einem Heap, vorgenommen werden müssen. So paßt DBIDA* die Gestalt des expliziten, d. h. des gespeicherten Teilgraphen, der äußeren Horizontlinie an und erst nach vielen Expansionen von Horizontknoten muß eine Änderung des gespeicherten Teilgraphen erfolgen. Dies steht im Gegensatz zu MA* und SMA*, die häufig für die Expansion eines weiteren Horizontknotens eine erhebliche Anzahl von teuren Update-Operationen, nämlich dem Einfügen und Entfernen in einer Vorwäertswarteschlange, durchfüren müssen.

Der andere Vorteil besteht darin, daß mehrfach auftretende Knoten im oberen Teil des Suchbaumes von DBIDA* erkannt werden. Dies führt zu einer wesentlich geringeren Anzahl zu expandierender Knoten bei einem konkreten Suchproblem, da i. a. Duplikate auftreten. Das Auftreten von Duplikaten ist ein in der Praxis nicht vernachlässigbares Phänomen, wie an dem Paradebeispiel der Einpersonenspiele, dem Fünfzehnerpuzzle, ersichtlich ist. Hier führen oft viele Zugfolgen zu ein und derselben Stellung. Werden Duplikate nicht erkannt, werden die Knoten der Teilbäume der Duplikate abwechselnd expandiert. Da diese häufig nicht in unmittelbarer Nachbarschaft voneinander liegen, werden von MA$^*$ beim Wechsel von einem Teilbaum zum anderen alle bereits gespeicherten Knoten wieder vergessen, was zu vielen vermeidbaren Knotenexpansionen und Heap-Updates führen kann. Dieser Nachteil besteht auch, zumindest tendenziell bei SMA$^*$. Solche Wechsel bedingen jedoch i. a. keine Veränderung des von DBIDA$^*$ gespeicherten Teilbaumes und folglich auch keine teueren Heap-Operationen.

# Literatur


[1] P. P. Chakrabarti, S. Ghose, A. Acharya and S. C. de Sarkar, *Heuristic search in restricted memory*, Artif. Intell. 41 (2) (1989), 197-221

[2] Ghosh, S., and Mahanti, A., *Bidirectional heuristic search with limited resources*, Inform. Process. Letters 40 (1991) 335-340

[3] R. E. Korf, *Depth-first iterative deepening: an optimal admissible tree search*, Artif. Intell. 27 (1) (1985), 97-109

[4] R. E. Korf, *Linear-space best-first search*, Artif. Intell. 62 (1993) 41-78

[5] A. Mahanti, D. S. Nau, S. Ghosh and L. N. Kanal, *An efficient iterative threshold heuristic tree search algorithm*, Tech. Report UMIACS TR92-29, CS TR 2853, Computer Science Department, University of Maryland, College Park, MD (1992)

[6] Nilsson, N., J., *Principles of Artificial Intelligenc*, Springer-Verlag, 1982

[7] Pearl, J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984

[8] Pearl, J., *Generalized best-first strategies and the optimality of $A^*$*, J. A. C. M. 32, 3 (July 1985), 505-536.

[9] V. N. Rao, V. Kumar and R. K. Korf, *Depth-first vs. best-first search*, in: Proceedings AAAI-91, Anaheim, CA (1991) 434-440

[10] S. Russell, *Efficient memory-bounded search methods*, in: Proceedings ECAI-92, Vienna, Austria (1992)

[11] U. K. Sarkar, P. P. Chakrabarti, S. Ghose and S. C. de Sarkar, em-pReducing reexpansions in iterative-deepening search by controlling cutoff bo unds, Artif. Intell. 50 (2) (1991) 207-221

[12] A. K. Sen and A. Bagchi, *Fast recursive formulations for best-first search that allow controlled use of memory*, in Proceedings IJCAI-89, Detroit, MI (1989) 297-302

[13] L. Taylor and R. E. Korf, *Pruning duplicate nodes in depth-first search*, in: Proceedings AAAI-93, Washington, DC (1993)

[14] B. W. Wah, *MIDA$^*$, an IDA$^*$ search with dynacmic control*, Tech. Report UILU-ENG-91-2216, CRHC-91-9, Center for Reliable and High-Performance Computing, Coordinated Science Laborartory, College of Engineering, University of Illinois at Urbana, Champaign-Urbana, IL (1991)