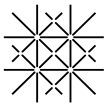


Universität
Basel

Symbolische Zustandsraumsuche mit Sentential Decision Diagrams

Bachelorarbeit

Philosophisch-Naturwissenschaftliche Fakultät der Universität Basel
Department Mathematik und Informatik
Künstliche Intelligenz
http://ai.cs.unibas.ch

Beurteiler: Prof. Dr. Malte Helmert
Zweitbeurteiler: Thomas Keller und Salome Eriksson

Simon Wallny
simon.wallny@stud.unibas.ch
2013-055-751

08. August 2016

Abstract

Auf dem Gebiet der Handlungsplanung stellt die symbolische Suche eine der erfolgversprechendsten angewandten Techniken dar. Um eine symbolische Suche auf endlichen Zustandsräumen zu implementieren bedarf es einer geeigneten Datenstruktur für logische Formeln. Diese Arbeit erprobt die Nutzung von *Sentential Decision Diagrams* (SDDs) anstelle der gängigen *Binary Decision Diagrams* (BDDs) zu diesem Zweck. SDDs sind eine Generalisierung von BDDs. Es wird empirisch getestet wie eine Implementierung der symbolischen Suche mit SDDs im FastDownward-Planer sich mit verschiedenen vtrees unterscheidet. Insbesondere wird die Performance von balancierten vtrees, mit welchen die Stärken von SDDs oft gut zur Geltung kommen, mit rechtsseitig linearen vtrees verglichen, bei welchen sich SDDs wie BDDs verhalten.

Inhaltsverzeichnis

1 Einführung

Viele Anwendungsgebiete der künstlichen Intelligenz lassen sich anhand einer endlichen Menge von Zuständen beschreiben, zwischen denen Übergänge existieren. Die Idee dabei ist, einen Aktionsplan zu finden, der einen von einem Startzustand über diese erlaubten Übergänge möglichst schnell zu einem gewünschten Zielzustand bringt. Ein Beispiel für ein solches Problem ist das *Wolf, Ziege und Kohlkopf*-Problem. Hierbei möchte man einen Weg finden einen Wolf, eine Ziege und einen Kohlkopf über einen Fluss zu bringen, kann aber nur einen von ihnen gleichzeitig transportieren und darf weder den Wolf mit der Ziege noch die Ziege mit dem Kohlkopf allein lassen. Eine Zustand ist hier dadurch gegeben, an welchem Ufer sich die Beteiligten befinden. Durch das Überqueren des Flusses – allein oder mit einem seiner Habseligkeiten – kann man von einem Zustand in den nächsten übergehen, solange der Übergang nicht die obigen Regeln verletzt, bis man im Zielzustand angelangt ist. Insbesondere interessiert man sich meist für den *günstigsten* (hier: kürzesten) Plan, falls mehrere existieren.

Nun ist es für derart kleine Probleme nicht schwer den gesamten Zustandsraum aufzuspannen und etwa mit dem Dijkstra-Algorithmus den kürzesten Pfad zu bestimmen. Doch da die Anzahl der möglichen Zustände oft exponentiell mit der Anzahl der Variablen steigt, die die Zustände bestimmen, wird der Zustandsraum sehr schnell zu groß um ihn im Speicher zu behalten. Stattdessen arbeitet man sich bei der Zustandsraumsuche vom Startzustand ausgehend schrittweise über seine Nachfolgerzustände vor bis man einen Zielzustand erreicht hat.

Sich auf diese Weise blind durch den Zustandsraum zu tasten kann sehr lange dauern und erfordern, einen zu großen Teil des Raumes im Speicher zu behalten, weshalb mehrere Ansätze bestehen, um die Suche zu verbessern. Einer der gängigsten darunter ist, mithilfe einer Heuristik – einer Funktion die schätzt wie nahe ein Zustand dem Zielzustand ist – zu bestimmen welche Zustände man zuerst betrachtet. Ein anderer Ansatz, welcher in dieser Thesis verwendet wird, ist die symbolische Suche. Hierbei werden, statt mit konkreten Zuständen zu arbeiten, ganze Mengen von Zuständen verarbeitet. Das erlaubt es einem viele Zustände parallel abzuarbeiten und so den Zustandsraum in großen Stücken zu erforschen. Mengen von Zuständen werden dabei mithilfe von aussagenlogischen Formeln beschreiben.

Dies erfordert eine Darstellung von logischen Formeln, die platzsparend ist und effizient

bestimmte Operationen wie Konjunktion und Disjunktion ermöglicht. Gängig ist, dafür *binary decision diagrams* (BDDs) zu verwenden. Hierbei werden, wie der Name vermuten lässt, um zu bestimmen ob ein Zustand von dem BDD beschrieben wird, Entscheidungen abhängig von der Belegung der (binären) Variablen gemacht, die in festgelegter Reihenfolge abgearbeitet werden. Die in dieser Thesis verwendeten *sentential decision diagrams* (SDDs) stellen eine Generalisierung der BDDs dar, welche mit Sätzen von Variablen arbeiten, wobei diese nicht in Reihenfolge sondern als Baum angeordnet sind.

2
Hintergrund

2.1 Zustandsraumsuche

Eine Probleminstanz der Zustandsraumsuche sei definiert als Tupel $\mathcal{S} = \langle S, A, cost, T, s_0, S_* \rangle$, wobei S eine endliche Menge von Zuständen und A eine endliche Menge von Aktionen ist. Desweiteren ist *cost* eine Funktion $A \to \mathbb{N}_0$ welche allen Aktionen Kosten zuweist. $T \subseteq S \times A \times S$ beschreibt zwischen welchen Zuständen mit welchen Aktionen Übergänge möglich sind. Eine Aktion $a \in A$ angewendet auf einen Zustand $s_1 \in S$ bewirkt einen Übergang in Zustand $s_2 \in S$ genau dann wenn $\langle s_1, a, s_2 \rangle \in T$. T ist dabei deterministisch in $\langle s_1, a \rangle$, es kann also keine zwei Tupel $\langle s_1, a, s_2 \rangle, \langle s_1, a, s_3 \rangle \in T$ geben, wenn nicht $s_2 = s_3$ gilt. $s_0 \in S$ ist der Startzustand und $S_* \subseteq S$ ist eine Menge von Zielzuständen. Der Zustand s' heißt *Nachfolger* von s, wenn mindestens ein Übergang von s zu s' existiert. Gesucht ist ein Plan $\pi = \langle a_1, ..., a_n \rangle$ mit $a_i \in A \; \forall i$ der eine Abfolge von Übergängen definiert, die einen ausgehend von einem Übergang $\langle s_0, a_1, s_1 \rangle \in T$ über n Schritte der Form $\langle s_{i-1}, a_i, s_i \rangle \in T$ vom Startzustand in einen Zielzustand bringt ($s_n \in S_*$). Die Kosten eines solchen Plans π sind $cost(\pi) = \sum_{i=1}^{n} cost(a_i)$. Ein Plan heißt *optimal* wenn kein Plan existiert dessen Kosten geringer sind.

2.2 STRIPS Formalismus

Der STRIPS Formalismus bietet eine formale Sprache zur Beschreibung solcher Probleminstanzen. [1] Eine STRIPS-Planungsaufgabe ist gegeben als 4-Tupel $\Pi = \langle V, A, I, G \rangle$. Hierbei existiert eine endliche Menge von binären Variablen V. Zustände sind als Variablenbelegungen für diese Variablen definiert (jede Variable $v \in V$ wird auf $\top$ (true) oder auf $\bot$ (false) abgebildet). Der Startzustand I bestimmt die initialen Variablenbelegungen. Die Zielbedingungen $G \subseteq V$ sind als Menge von Variablen gegeben, die ein Zustand auf true abbilden muss um als Zielzustand zu gelten. Die Belegung der übrigen Variablen ist dabei irrelevant. Aktionen schließlich sind bestimmt durch ihre Kosten $cost(a) \in \mathbb{N}_0$ und jeweils drei Untermengen von V: Preconditions – Variablen, die ein Zustand auf true abbilden muss bevor die Aktion angewendet werden kann – sowie *add*- und *delete*-Effekte, welches Variablen sind, die durch das Anwenden der Aktion respektive auf true und auf false gesetzt werden.

2.3 Explizite Suche

Bei der expliziten Suche arbeitet man sich schrittweise vom Startzustand ausgehend von Nachfolger zu Nachfolger durch den Zustandsraum bis ein Zielzustand gefunden ist. Zu diesem Zwecke definiert man sich eine *open list* – eine Liste von Suchknoten $N = \langle s, g, a, N_{pre} \rangle$ welche einen Zustand s, die zum Erreichen des Zustandes akkumulierten Kosten g sowie einen Vorgängerknoten N_{pre} und eine zugehörige Aktion a beinhalten. Man füllt diese Liste zu Beginn mit einem Knoten für den Startzustand, Kosten 0, keiner Aktion und keinem Vorgängerknoten. Nun nimmt man sukzessive Knoten aus dieser Liste und *expandiert* sie. Das bedeutet, dass man sie aus der Liste entfernt und dafür Knoten für alle Nachfolger von s einfügt. Diese Nachfolger-Knoten haben den expandierten Knoten als Vorgänger und addieren zu dessen Kosten noch die Kosten der Aktion, die zu dem Übergang führt – welche sie ebenfalls speichern. Wenn man immer den Knoten mit den geringsten Kosten g aus der open list entfernt (Uniform Cost Search) kann man sichergehen, dass wenn der Zustand eines zu expandierender Knotens ein Zielzustand ist, der Plan, der sich nun aus der Aneinanderreihung der Aktionen aller Vorgängerknoten rekursiv bis zum Startzustand ergibt, optimal ist.

3
Symbolische Suche

Die symbolische Suche arbeitet nicht mit expliziten Zuständen sondern mit *Mengen von Zuständen*, welche durch aussagenlogische Formeln beschrieben sind. Hat man beispielsweise die Variablen $V = \{a, b, c\}$ und die Zustände $\{a \mapsto \top, b \mapsto \bot, c \mapsto \top\}$ sowie $\{a \mapsto \bot, b \mapsto \bot, c \mapsto \top\}$ wird die Menge beider Zustände durch die Formel $(((a \vee \neg a) \wedge \neg b) \wedge c)$ beschrieben. Aktionen sind Operatoren die auf eine Menge von Zuständen angewandt die Menge aller durch diese Aktion hervorgerufenen Nachfolger der Zustände liefern. Die open list beinhaltet hierbei statt expliziten Knoten nur Mengen von Zuständen S_g die mit bestimmten Kosten g vom Startzustand aus erreicht werden können. Da die Abfolge von Aktionen, die zu einem bestimmten Zustand in der Menge geführt hat, nicht nachvollziehbar ist, ist die Planrekonstruktion sobald ein Zielzustand gefunden ist komplizierter als bei der expliziten Suche. Als "gefunden" gilt ein Zielzustand sobald man eine Menge S_g expandiert die einen nichtleeren Schnitt mit der Menge von Zielzuständen S_* hat. Folgt man dem Uniform Cost Search-Prinzip, immer die Menge mit dem kleinsten g-Wert zu expandieren, sind die Kosten der Menge S_g, die die Zielzustände schneidet, die Kosten eines optimalen Plans.

Der in Algorithmus 3.1 gegebene Pseudo-Code beschreibt einen Ansatz zur Implementierung von symbolischer Suche nach dem Uniform Cost Search-Prinzip, angelehnt an die Beschreibung von Álvaro Torralba[2]. Die *image* Funktion wird im nächsten Abschnitt erklärt. Anzumerken ist, dass dieser Algorithmus nur funktioniert, wenn keine Aktionen mit Nullkosten existieren. Es ist möglich den Algorithmus mit diesen verträglich zu machen, indem man vor jeder Expansion von $open_{g_{min}}$ zuerst alle Aktionen mit Nullkosten separat bearbeitet und die so erreichbaren Zustände in $open_{g_{min}}$ einträgt.

Transitionen

Um Aktionen in der symbolischen Suche verwenden zu können bedarf es einer Methode sie auf eine *Menge* von Zuständen anzuwenden. Das bedeutet es muss eine Funktion $S_2 = image(S_1, T_a)$ gegeben sein, wobei S_2 die Nachfolger aller Zustände in S_1 enthält, die durch die in Anwendung der Aktion a erfolgen. Insbesondere werden Zustände $s \in S_1$ für die kein Übergang mit Aktion a möglich ist keine Nachfolger in S_2 generieren. Dies kann unter Zuhilfenahme von Hilfsvariablen getan werden. Das sei an folgendem Beispiel illustriert:

Algorithmus 3.1: **Pseudo-Code für Uniform-Cost Symbolische Suche:**

```
Input: STRIPS–Planungsaufgabe Π = ⟨V, I, G, A⟩
Output: Optimaler Plan π oder "unlösbar"
open_0 = {s_0}
for each a ∈ A do
  T_a = erstelleTransition(a)
  T = T ∪ {T_a}
end
while 'open nicht leer' do
  g_min = min{g : open_g ≠ ∅}'
  closed_{g_min} = open_{g_min}
  if open_{g_min} ∩ S_* ≠ ∅ then
    return Planrekonstruktion(open_{g_min} ∩ S_*, g_min, T, closed)
  end
  for each T_a ∈ T do
    c = cost(a)
    open_{g_min+c} = open_{g_min+c} ∪ image(open_{g_min}, T_a)
  end
  open_{g_min} = ∅
end
return "unlösbar"
```

Gegeben sei ein STRIPS-Planungsproblem mit Variablen $V = \{L, R, P\}$ welche für zwei Orte Links und Rechts sowie ein aufhebbares Paket stehen. Ich werde im Folgenden als Kurzschreibweise $\{A, \neg B\}$ für einen Zustand $\{A \mapsto \top, B \mapsto \bot\}$ nutzen.
Nun hat man eine Menge von Zuständen S_1. Diese Menge enthält 3 Zustände, $\{L, \neg R, \neg P\}$, $\{L, \neg R, P\}$ und $\{\neg L, R, \neg P\}$. Als Aussagenlogische Formel ausgedrückt heißt das $(L \wedge \neg R) \vee (\neg L \wedge R \wedge \neg P)$. Auf diese Menge möchte man die Aktion "gehe nach R" anwenden. Formal hat diese Aktion Preconditions $pre(a) = \{L\}$, Add-Effekte $add(a) = \{R\}$ und Delete-Effekte $del(a) = \{L\}$.
Da diese Aktion nur auf die ersten beiden Zustände in S_1 anwendbar ist und bei diesen schlicht die Belegung von L und R vertauscht ist leicht ersichtlich, dass $S_2 = \{\{\neg L, R, \neg P\}, \{\neg L, R, P\}\}$ das gewünschte Endergebnis ist, was der Formel $(\neg L \wedge R)$ entspricht.

Um dorthin zu gelangen definiert man sich zunächst für jede Variable in $v \in V$ eine Hilfsvariable v', deren Zustand später den Zustand ihrer Variable in den Nachfolgerzuständen bestimmt. Die Transition T_a für die Aktion kann nun selbst als aussagenlogische Formel definiert werden. Dazu folgt man folgendem Schema:

$$T_a = \bigwedge_{p \in pre(a)} (p) \wedge \bigwedge_{b \in add(a)} (b') \wedge \bigwedge_{d \in del(a)} (\neg d') \wedge \bigwedge_{x \notin (add(a) \cup del(a))} (x' \leftrightarrow x)$$

Diese Transition T_a wird nun verwendet indem man die Konjunktion mit der Formel für S_1 bildet. Die vier Teile der Transition haben dabei – in Reihenfolge – folgende Effekte:

1. Nur die Zustände die die Precondition erfüllen werden berücksichtigt. Würde kein einziger Zustand aus S_1 die Precondition erfüllen wäre der Schnitt mit nur diesem Teil

bereits leer.

2. Jede Variable die durch die Aktion auf true gesetzt wird bekommt ihre jeweilige Hilfsvariable zur Formel hinzugefügt (was bedeutet, dass jeder Zustand in S_2 diese Variable notwendigerweise auf true setzen muss).

3. Analog; jede Variable die durch die Aktion auf false gesetzt wird bekommt ihre jeweilige Hilfsvariable in negierter Form zur Formel hinzugefügt (was bedeutet, dass jeder Zustand in S_2 diese Variable notwendigerweise auf false setzten muss).

4. Jede Variable die weder in Add- noch in Delete-Effekten vorkommt (deren Belegung also nicht von der Aktion abhängt sondern davon, wie sie in dem Zuständen in S_1 belegt war) bekommt ihre Hilfsvariable hinzugefügt, die genau unter denselben Umständen true oder false ist wie in S_1.

Diese Teile schrittweise auf unser Beispiel angewendet sieht folgendermaßen aus:

1. $((L \wedge \neg R) \vee (\neg L \wedge R \wedge \neg P)) \wedge (\bigwedge_{p \in pre(a)} (p))$
   $= ((L \wedge \neg R) \vee (\neg L \wedge R \wedge \neg P)) \wedge (L)$
   $= (L \wedge \neg R)$.
   Das beschreibt die folgenden Zustände: $\{L, \neg R, P\}$ und $\{L, \neg R, \neg P\}$ – also genau die beiden Zustände in S_1 welche die Precondition erfüllen.

2. $(L \wedge \neg R) \wedge (\bigwedge_{b \in add(a)} (b'))$
   $= (L \wedge \neg R) \wedge (R')$
   $= (L \wedge \neg R \wedge R')$
   Wie gesagt sind es die Hilfsvariablen die die Nachfolger beschreiben. Wir wissen also schon mal, dass alle Nachfolger R auf true setzen – was stark zu erwarten ist, wenn man eine Aktion namens "nach R gehen" anwendet.

3. $(L \wedge \neg R \wedge R') \wedge (\bigwedge_{d \in del(a)} (\neg d'))$
   $= (L \wedge \neg R \wedge R') \wedge (\neg L')$
   $= (L \wedge \neg R \wedge R' \wedge \neg L')$
   Nun ist $\neg L$ als notwendige Bedingung hinzugekommen. Für die Nachfolger bedeutet dass, dass sie ausnahmslos L auf false setzen – sinnig, da sie sich dort nach dem bewegen zu R nicht mehr aufhalten können.

4. $(L \wedge \neg R \wedge R' \wedge \neg L') \wedge (\bigwedge_{x \notin (add(a) \cup del(a))} (x \leftrightarrow x'))$
   $= (L \wedge \neg R \wedge R' \wedge \neg L') \wedge (P' \leftrightarrow P)$
   $= (L \wedge \neg R \wedge R' \wedge \neg L' \wedge ((P' \wedge P) \vee (\neg P' \wedge \neg P)))$
   Da das Paket P von der Bewegung nach R nicht beeinflusst wird kommt nun ein Teil mit der Bedeutung "P genau wie es vorher war" hinzu. Da P nicht in der vorherigen Formel vorkam (man konnte das Paket haben oder auch nicht) bleibt dieser Teil einfach so stehen.

Als nächstes vergisst man in der resultierenden Formel alle Nicht-Hilfsvariablen, denn es bestimmen wie erwähnt nur diese die Nachfolger. Formal bedeutet "vergessen" eine Existenzquantifizierung. Das Resultat unserer Beispielformel lautet wie folgt:
$(R' \wedge \neg L' \wedge ((P') \vee (\neg P')))$
$= (R' \wedge \neg L')$
Da der Zustand von P vorher beliebig war fällt er auch jetzt wieder weg. Alles was jetzt noch zu tun bleibt ist alle Hilfsvariablen zu ihren originalen Gegenstücken umzubenennen um wieder im alten System zu sein und man hat wie erwünscht die Formel $(\neg L \wedge R)$ was der Menge $S_2 = \{\{\neg L, R, \neg P\}, \{\neg L, R, P\}\}$ entspricht.

Eine besondere Eigenschaft dieser Art, Übergänge zu berechnen, ist, dass sie sich leicht umkehren lässt, um alle möglichen Vorgänger einer Zustandsmenge zu bestimmen. Sei nun S_2 gegeben und S_1 zu berechnen geht man einfach in umgekehrter Reihenfolge vor: Man benennt alle Variablen in S_2 zu Hilfsvariablen um, schneidet mit der gewünschten Transition T_a wonach man stattdessen alle Hilfsvariablen vergisst und nur die Originalvariablen behält, die durch den Schnitt in der Formel auftauchen. Diese Methode sei $S_1 = invimage(S_2, T_a)$ genannt.

Planrekonstruktion

Um den genauen Plan zu rekonstruieren muss man während der Suche alle bereits expandierten Zustandsmengen in einer sogenannten *closed list*, die wie die open list für jeden Kostenwert g eine Menge von Zuständen enthält, welche mit Kosten g erreicht werden können, speichern. Durch umgekehrte Anwendung der Aktionen auf die gefundenen Zielzustände lassen sich mögliche Vorgänger generieren. Sind Vorgänger für eine Aktion mit Kosten c in der closed list mit Kosten $g - c$ abgelegt, so hat man einen Zustand gefunden durch den ein optimaler Plan führt, und kann von dort aus weiter suchen bis man zum Startzustand gelangt.

Die Logik hinter diesem Vorgehen ist, dass wenn man weiß, dass ein Zielzustand mit Kosten g_* optimal erreichbar ist und man ferner weiß, dass ein Zustand s existiert, von dem aus man mit Kosten c zum Zielzustand gelangt wobei dieser Zustand selbst mit Kosten $g_* - c$ vom Start aus erreichbar ist, dann führt mindestens ein optimaler Plan durch diesen Zustand s.

Der in Algorithmus 3.2 gegebene Pseudo-Code beschreibt einen Ansatz zur Implementierung der Planrekonstruktion, erneut angelehnt an Torralba's Thesis[2]. Dieser Algorithmus geht ebenfalls davon aus, dass keine Aktionen mit Nullkosten bestehen. Um mit diesen verträglich zu sein muss aber lediglich als Abbruchbedingung statt $g = 0$ geprüft werden ob es sich um den Startzustand handelt ($S \cup s_0$), welcher zusätzlich als Parameter gegeben sein muss (da es bei $g = 0$ sein kann dass man erst einen Zustand gefunden hat, in den man mit kostenfreien Aktionen vom Startzustand aus kommt).

Algorithmus 3.2: **Pseudo-Code für Planrekonstruktion:**

```
Input: Gefundene Zielzustände S_Ziel
Input: Optimale Kosten g_*
Input: Transitionen T
Input: Closed List closed
Output: Optimaler Plan π = [a_1, ..., a_n]
π = []
g = g_*
S = S_Ziel
while g > 0
  for each T_a ∈ T
    S' = invimage(S, T_a)
    c = cost(a)
    if S' ∩ closed_{g-c} ≠ ∅ then
      S = S' ∩ closed_{g-c}
      g = g - c
      π = a||π
      break
    end
  end
end
```

In Abbildung 3.1 sieht man eine Visualisierung der Zustandsmengen zum Zeitpunkt der Planrekonstruktion und des Plans der sich durch diese hindurchzieht. Einträge der closed list für bestimmte g-Werte sind nicht zwangsläufig disjunkt und nichtleer. Die Planrekonstruktion muss nicht durch alle g-bins gehen.

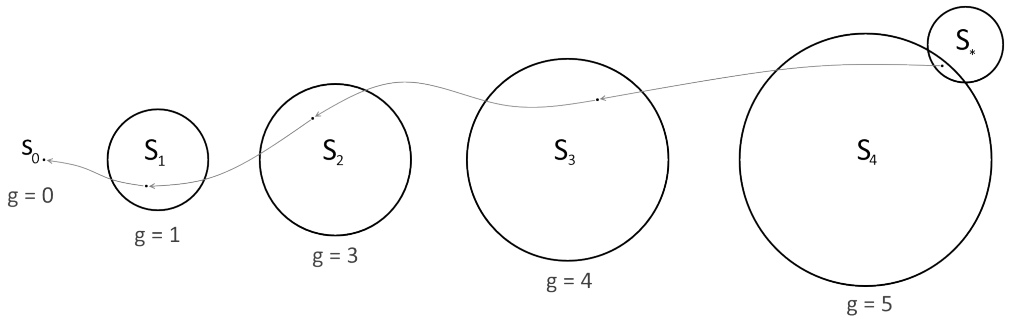


Abbildung 3.1: Visualisierung der Planrekonstruktion durch die closed list.

4

Sentential Decision Diagrams

Sentential Decision Diagrams (SDDs)[3] sind eine Darstellung von aussagenlogischen Formeln, die einige positive Eigenschaften besitzt. SDDs sind kanonisch, haben in ihrer Größe bekannte Obergrenzen die so gut oder besser als gängige Darstellungen sind, und bieten Konjunktion und Disjunktion in polynomieller Laufzeit.
SDDs stellen eine Erweiterung der gängigeren Binary Decision Diagrams (BDDs) dar. Während BDDs bei einer festen Variablenordnung kanonisch sind, sind Variablen in SDDs nicht in Abfolge, sondern als binärer Baum, einem sogenannten *vtree*, geordnet. In dem Sonderfall dass der vtree rechtsseitig linear ist, also nur die rechten Knoten des Baumes Kindknoten haben, verhält sich ein SDD genau wie ein BDD.

4.1 BDDs

Binary Decision Diagrams (BBDs)[4] bestehen aus Entscheidungen basierend auf der Belegung von (binären) Variablen. Diese werden in Reihenfolge abgearbeitet. In Abbildung 4.1 ist als Beispiel ein BDD für die aussagenlogische Formel $(A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ gegeben. Die Variablenordnung ist hierbei einfach ABCD. Um nun zu testen ob eine Variablenbelegung mit dem BDD konform ist beginnt man mit dem Knoten A. Ist A true, folgt man dem durchgezogenen Pfeil, ist A false, folgt man dem gestrichelten Pfeil. So arbeitet man sich weiter bis man zum Ergebnis $\top$ oder $\bot$ – konform oder nicht konform – kommt.

4.2 SDDs

Bei Sentential Decision Diagrams wird – anders als bei BDDs – eine Entscheidung nicht aufgrund einer einzigen Variablenbelegung getroffen sondern aufgrund eines ganzen Satzes. In jeder Entscheidung partitioniert man den Raum anhand einiger Variablen, was bedeutet, dass für die Entscheidung mehrere Formeln über eine Untermenge von Variablen gegeben sind, von denen für jede mögliche Variablenbelegung genau eine erfüllt ist. Diese Formeln heißen *primes*. Zu jeder dieser primes kommt eine zweite Formel, genannt *sub*, die beschreibt, was in dem Fall, in dem diese prime erfüllt ist, für die anderen Variablen gelten muss. Sowohl primes als auch subs werden, solange sie von mehreren Variablen abhängen, wieder

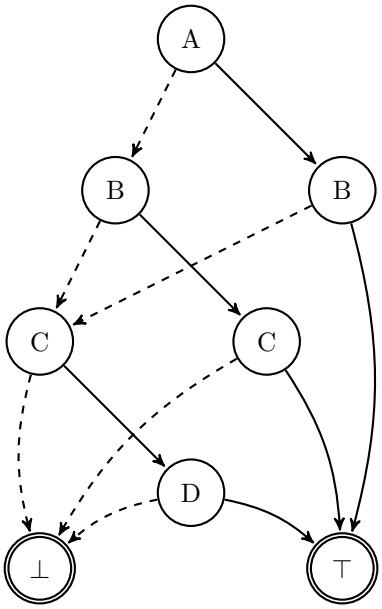


Abbildung 4.1: BDD für die Formel $(A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$

als SDDs beschrieben, wodurch ein Baum entsteht wie er in Abbildung 4.2 für die Formel $(A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ gegeben ist. Jedes Teilbaum-SDD kennt dabei nur eine Untermenge der Variablen seines Eltern-SDDs.

Möchte man jetzt beispielsweise die Variablenbelegung $I = \{A \mapsto \bot, B \mapsto \bot, C \mapsto \top, D \mapsto \top\}$ auf Konformität mit dem Beispiel-SDD aus Abbildung 4.2 prüfen (also darauf, ob es die Formel $(A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ erfüllt), geht man wie folgt vor. Man beginnt mit dem Wurzelknoten. Hier partitioniert man anhand von drei primes, nennen wir sie prime1 bis prime3. Zuerst will man sehen ob prime1 erfüllt ist. Da prime1 selbst wieder ein SDD ist sieht man sich dieses an. Hier hat man zwei primes, B und $\neg B$. B ist nicht erfüllt, aber $\neg B$ ist es. Das bedeutet um mit dem SDD prime1 konform zu sein müsste man den korrespondierenden sub $\bot$ erfüllen, was trivialerweise nicht der Fall ist. Da prime1 nicht erfüllt ist schaut man sich als nächstes prime2 an. Auch hier wieder müsste, da $\neg B$ erfüllt ist auch $\bot$ erfüllt sein was unmöglich ist. Also schaut man sich zu guter letzt prime3 an, was nun erfüllt sein muss da es keine Alternativen mehr gibt. Und tatsächlich, prime3 $= \neg B$ ist erfüllt. Demnach müssen die verbleibenden Variablen den sub erfüllen, welcher wieder ein verschachteltes SDD ist. Von dessen primes ist D erfüllt, was bedeutet dass dessen sub C erfüllt sein muss. Das ist der Fall, also ist die Variablenbelegung mit dem SDD konform.

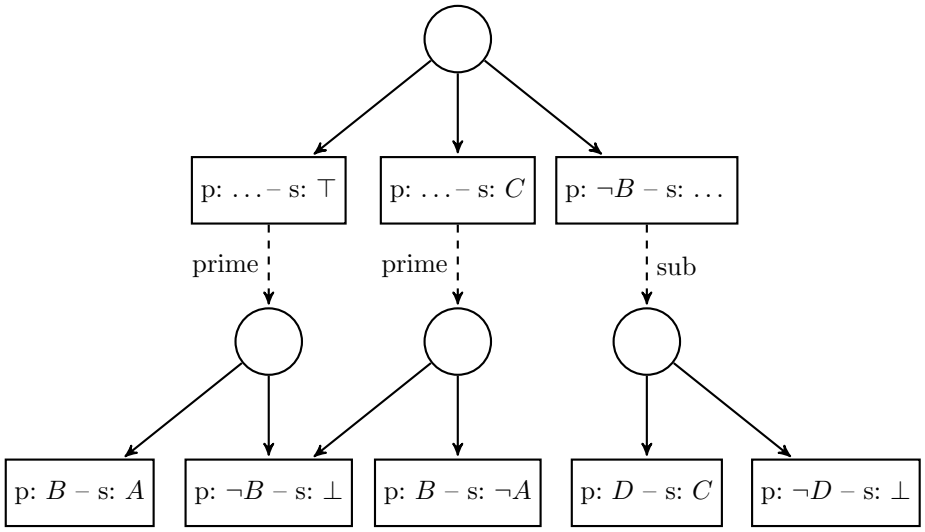


Abbildung 4.2: SDD für die Formel $(A \wedge B) \vee (B \wedge C) \vee (C \wedge D)$ aus Darwiche's Thesis[3]

5
Experimente

Ich habe für diese Thesis eine Implementierung von symbolischer Suche aufbauend auf den bestehenden Fast Downward Planer[5] geschrieben, die das SDD Package von Arthur Choi und Adnan Darwiche[1] benutzt.

SDDs dienten dabei sowohl der Kodierung von Mengen von Zuständen als auch der Anwendung der Übergänge. Das Programm wurde anschließend dafür benutzt die Performanz von SDDs mit verschiedenen vtrees zu vergleichen. Die Variablenordnung ist dabei für alle Implementationen gleich, sie unterscheiden sich lediglich in der Struktur der Bäume. Die vier verwendeten Baumstrukturen sind in Abbildung 5.1 bis 5.4 illustriert.

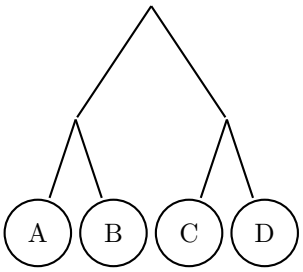


Abbildung 5.1: balanced vtree

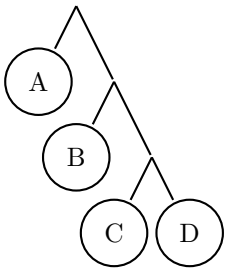


Abbildung 5.2: right linear vtree

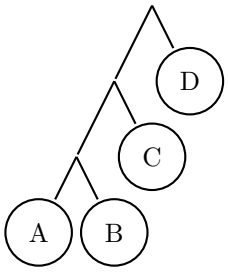


Abbildung 5.3: left linear vtree

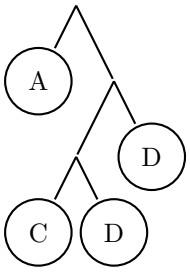


Abbildung 5.4: vertical vtree

[1] http://reasoning.cs.ucla.edu/sdd/

Sämtliche Experimente wurden mit Constraints von 2GB Speicher und 30 Minuten maximaler Laufzeit ausgeführt. Als Testinstanzen wurde die für FastDownward am häufigsten verwendete Suite benutzt[2], von der diejenigen Domänen ausgewählt wurden, die STRIPS-Probleme ohne Nullkosten beschreiben.

5.1 Coverage

Wie viele Probleminstanzen aus den Testdomänen die verschiedenen Implementationen unter den oben angegebenen Constraints lösen konnten ist in der unten angegebenen Tabelle ersichtlich. Insbesondere fällt auf, dass die rechtslinearen vtrees – die BDDs entsprechen – fast nie besser abschneiden als die anderen vtrees. Insbesondere balancierte vtrees zeigen einen Trend an, besser abzuschneiden, wobei auch diese nicht in allen Situationen am besten geeignet sind.

[2] https://bitbucket.org/aibasel/downward-benchmarks

Domain	Total	Balanced	Right Linear	Left Linear	Vertical
airport	50	**11**	**11**	**11**	**11**
barman-opt11-strips	20	0	0	0	0
blocks	35	**10**	**10**	9	9
depot	22	**2**	**2**	1	1
driverlog	20	**4**	2	3	3
elevators-opt08-strips	30	0	0	0	0
elevators-opt11-strips	20	0	0	0	0
floortile-opt11-strips	20	0	0	0	0
freecell	80	**6**	3	5	2
grid	5	**1**	0	0	**1**
gripper	20	**8**	6	6	7
logistics00	28	**10**	**10**	9	**10**
logistics98	35	0	0	0	0
miconic	150	49	**50**	49	49
movie	30	**30**	**30**	**30**	**30**
mprime	35	**5**	3	3	3
mystery	30	**7**	**7**	**7**	**7**
nomystery-opt11-strips	20	**5**	3	4	4
openstacks-strips	30	6	6	**7**	6
parking-opt11-strips	20	0	0	0	0
pathways-noneg	30	**4**	**4**	**4**	**4**
pipesworld-notankage	50	**5**	3	3	3
pipesworld-tankage	50	2	2	**3**	2
psr-small	50	47	47	**48**	47
rovers	40	**5**	4	**5**	**5**
satellite	36	3	3	**4**	3
scanalyzer-08-strips	30	**5**	4	3	3
scanalyzer-opt11-strips	20	**2**	**2**	1	1
storage	30	7	7	7	**9**
tidybot-opt11-strips	20	**1**	**1**	**1**	**1**
tpp	30	**5**	**5**	**5**	**5**
transport-opt08-strips	30	**6**	**6**	**6**	5
transport-opt11-strips	20	**1**	**1**	**1**	0
trucks-strips	30	**3**	2	**3**	2
visitall-opt11-strips	20	**8**	7	**8**	**8**
woodworking-opt08-strips	30	**4**	**4**	3	**4**
woodworking-opt11-strips	20	0	0	0	0
zenotravel	20	**6**	5	5	5
Sum	1256	**268**	250	254	250

5.2 Speicher und Laufzeit

In sowohl Speicherverbrauch als auch Laufzeit schnitten in den Tests SDDs mit balancierten vtrees im Schnitt am besten ab, während SDDs mit rechtsseitig linearen vtrees am schlechtesten abschnitten. Auf den Probleminstanzen die von allen Implementationen gelöst wurden benötigten balancierte SDDs im Vergleich zu rechtsseitig linearen in der Summe 80.28% des Speichers und im arithmetischen Mittel 63.42% der Laufzeit.

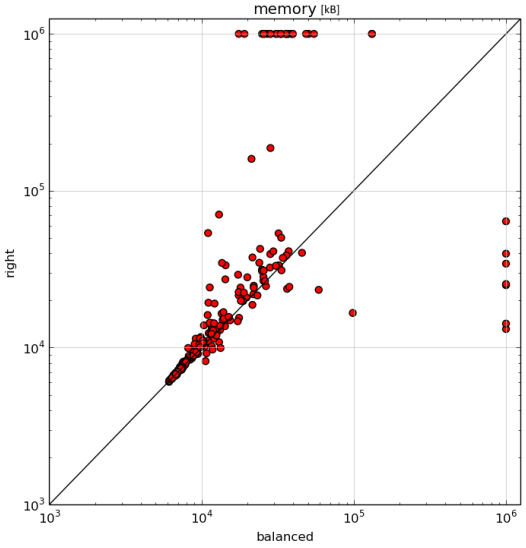


Abbildung 5.5: Speicher

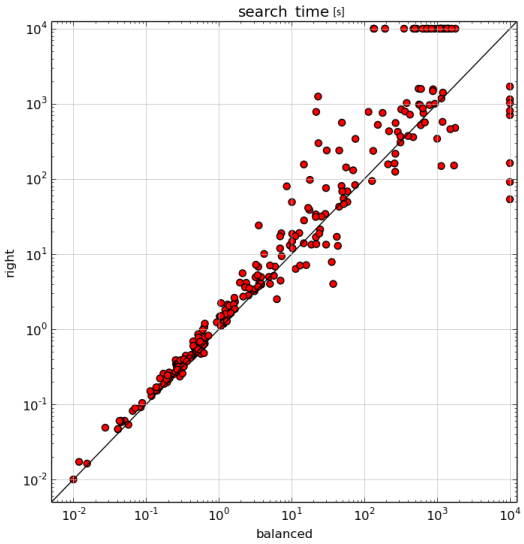


Abbildung 5.6: Laufzeit

6
Schlussfolgerung

Da SDDs im Vergleich zu BDDs deutlich jünger sind, sind sie noch nicht annähernd so gut auf theoretischer Ebene erforscht, wenngleich was bekannt ist vielversprechend wirkt. Aus den hier erzeugten empirischen Ergebnissen ist allerdings ein deutlicher Trend ersichtlich, wonach SDDs gegenüber BDDs in der Praxis besser abschneiden. Die Verwendung von SDDs statt BDDs stellt dabei wenig zusätzlichen Aufwand dar. – die Benutzung bei der Implementierung einer Anwendung, etwa eines Planers, ist annähernd identisch. Der einzige Teil der sich hierbei unterscheidet ist die Wahl eines geeigneten vtrees anstelle einer einfachen Variablenordnung. Da allerdings selbst die Wahl eines einfachen balancierten Baumes aufgrund einer Variablenordnung die Performance sichtlich verbessert ist dies kein Hinderungsgrund. Vielmehr präsentiert sich die Optimierung des vtrees als noch weitgehend unerforschtes Gebiet mir vielversprechenden Aussichten an.

Ein Ansatz zur Bestimmung eines guten vtrees, der interessante Ergebnisse verspricht, wäre beispielsweise Variablen anhand eines Causal Graphs zu ordnen, sodass Variablen, die sich stark beeinflussen, nahe aneinanderliegen.

Literaturverzeichnis

[1] Fikes, R. and Nilsson, N. J. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artifcial Intelligence*, 2(3-4):189–208 (1971).

[2] Torralba, Á. *Symbolic Search and Abstraction Heuristics for Cost-Optimal Planning*. Ph.D. thesis, Universidad Carlos III de Madrid (2015).

[3] Darwiche, A. SDD: A New Canonical Representation of Propositional Knowledge Bases. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence (IJCAI)*, pages 819–826 (2011).

[4] Bryant, R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35(8):677–691 (1986).

[5] Helmert, M. The Fast Downward Planning System. *Journal of Artifcial Intelligence Research*, 26:191–246 (2006).

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

includes Declaration on Plagiarism and Fraud
beinhaltet Erklärung zu Plagiat und Betrug

Author — Autor
Simon Wallny

Matriculation number — Matrikelnummer
2013-055-751

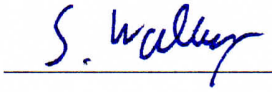
Title of work — Titel der Arbeit
Symbolische Zustandsraumsuche mit Sentential Decision Diagrams

Type of work — Typ der Arbeit
Bachelorarbeit

Declaration — Erklärung
I hereby declare that this submission is my own work and that I have fully acknowledged the assistance received in completing this work and that it contains no material that has not been formally acknowledged. I have mentioned all source materials used and have cited these in accordance with recognised scientific rules.

Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe. Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Basel, 08. August 2016



Signature — Unterschrift