Technische Universität Chemnitz

Fakultät für Informatik

CSR-98-01

Workshop über Komplexitätstheorie, Datenstrukturen und effiziente Algorithmen

Tagungsband zum 34. Workshop

Andreas Goerdt (Ed.)

10. März 1998 TU Chemnitz

Chemnitzer Informatik-Berichte

Chemnitzer Informatik-Berichte

In der Reihe der Chemnitzer Informatik-Berichte sind folgende Berichte erschienen:

CSR-95-01	25 Jahre Informatik – Akademischer Festakt am 4.November 1994, Januar 1995
CSR-95-02	W.Rehm u.a., Parallelrechner und Parallelprogrammierung, April 1995
CSR-95-03	H.Schreiter, Modellierung technischer Objekte und Anwendungen, Mai 1995
CSR-95-04	P.Horster, M.Michels, H.Petersen, Zertifikate und Konzepte selbstzertifizierender Schlüssel, Juli 1995
CSR-95-05	P.Horster, M.Michels, H.Petersen, Meta-Message Recovery and Meta-Blind signature schemes and their applications, Juli 1995
CSR-95-06	P.Köchel, Performance Measure Properties for Finite Queueing Systems, August 1995
CSR-95-07	S.Graupner, Nichtprozedurale Ablaufformen in imperativen Sprachen – Coroutinen und preemptive Threads in C –, August 1995
CSR-95-08	M.Jungmann, Wombat - Ein System zur Bewertung maschineller Lernverfahren hinsichtlich ihrer Klassifikationsgüte, August 1995
CSR-96-01	W.Benn, Y.Chen, I.Gringer, A Rule-based Strategy for Schema Integration in a Heterogeneous Information Environment, Januar 1996
CSR-96-02	W.Benn, I.Gringer, Datenbank-Anwendungen über das Internet, Februar 1996
CSR-96-03	W.Kalfa, Dynamische Adaption in Betriebssystemen – Das CHEOPS-Projekt, März 1996
CSR-96-04	Jahresbericht der Fakultät für Informatik 1995, Januar 1996
CSR-96-05	D.Monjau (Hrsg.), Custom Computing - GI/ITG Workshop, Juni 1996, Schloß Dagstuhl
CSR-96-06	W.Dilger, M.Schlosser, J.Zeidler, A.Ittner, Beiträge zum 9. Fachgruppentreffen Maschinelles Lernen der GI-Fachgruppe 1.1.3
CSR-97-01	Jahresbericht der Fakultät für Informatik 1996, Januar 1997
CSR-97-02	D.Monjau (Hrsg.), Hardwarebeschreibungssprachen und Modellierungsparadigmen, 3. ITG/GI/GMM–Workshop, 26.–28. Februar 1997, Holzhau
CSR-97-03	Y.Chen, W.Benn, A Systematic Method for Query Evaluation in Federated Relational Databases, Juni 1997
CSR-97-04	A.Goerdt, The Giant Component Threshold for Random Regular Graphs with Edge Faults, Juni 1997

Zeit	Referent	Thema
09:30 - 09:55	Kaffee	
09:55 - 10:00	Begrüßung	
10:00 - 10:30	Thomas Schwentick	Lokalität ordnungsinvarianter Formeln
10:30 - 11:00	Norbert Klasner	Structural Results about Exact Learning with Unspecified Attribute Values
11:00 - 11:30	Peter Damaschke	Attribut-effizientes Lernen
11:30 - 12:00	Arfst Nickelsen	Welche polynomiellen $\mathcal{D}$-Verboseness-Klassen enthalten p-bi-immune Mengen?
12:00 - 13:30	Mittagspause	
13:30 - 14:00	Christian Stangier	OBDD-based Verification of Communication Protocols – Methods for the Verification of Data Link Protocols
14:00 - 14:30	Harald Hempel	Translating Equality Downwards
14:30 - 15:00	Stefan Edelkamp	Lernen von Sackgassen in Sokoban
15:00 - 15:30	Christian Schindelhauer	Computational Error Complexity Classes
ca. 15:30	Ende des Workshops	

Martin Grohe
Uni Freiburg

Thomas Schwentick
Uni Mainz

Durch Arbeiten von Libkin und anderen Autoren hat sich in den letzten beiden Jahren die Hoffnung ergeben, die Komplexitätsklassen TC^0 und LOGSPACE könnten durch den Nachweis, daß TC^0 nur lokale Eigenschaften umfaßt, voneinander getrennt werden. Grob gesagt heißt dabei eine Eigenschaft von Tupeln $\overline{x}$ lokal, wenn sie nur vom Isomorphietyp einer Umgebung von $\overline{x}$ abhängt. LOGSPACE enthält hochgradig nicht-lokale Eigenschaften, wie z.B. (s,t)-Erreichbarkeit.

Obwohl inzwischen von Hella und Grohe gezeigt wurde, daß TC^0 nicht im strengen Sinne (mit konstant großen Umgebungen) lokal ist, ist es nicht ausgeschlossen, daß TC^0 in einem schwächeren Sinne lokal ist. In der dem Vortrag zugrundeliegenden Arbeit wird gezeigt, daß alle Eigenschaften, die von ordnungsinvarianten First-Order Formeln ausgedrückt werden können (einer Teillogik der TC^0 charakterisierenden Logik), lokal im strengen Sinne sind. Damit wird eine offene Frage von Libkin beantwortet.

Attribute Values

Andreas Birkendorf, Norbert Klasner, Christian Kuhlmann and Hans U. Simon

Lehrstuhl Mathematik und Informatik
Fakultät für Mathematik
Ruhr-Universität Bochum
D-44780 Bochum

This paper deals with the UAV learning model of Goldman, Kwek and Scott [2], where "UAV" is the acronym for "Unspecified Attribute Values". As in [2], we consider exact learning within the UAV framework. A smooth transition between exact learning in the UAV setting and standard exact learning is obtained by putting a fixed bound r on the number of unspecified attribute values per instance. For $r = 0$, we obtain the standard model. For $r = n$ (the total number of attributes), we obtain the (unrestricted) UAV model. Between these extremes, we find the hierarchies $(\text{UAV-MQ}_r)_{0 \le r \le n}$, $(\text{UAV-EQ}_r)_{0 \le r \le n}$, and $(\text{UAV-ARB-EQ}_r)_{0 \le r \le n}$.

Our main results are as follows. We present various lower bounds on the number of ARB-EQs and UAV-MQs in terms of the Vapnik Chervonenkis dimension of the concept class. We show furthermore that a natural extension of Angluin's Sunflower Lemma [1] is still applicable in the exact UAV learning model. Our *UAV Sunflower Lemma* allows to establish exponentially large lower bounds on the necessary number of UAV-MQs for several popular concept classes. On the other hand, we can show that slight simplifications of these classes are efficiently learnable using only few UAV-MQs. Finally, we investigate the inherent structure of the aforementioned three hierarchies and the relations between them. It turns out that query type UAV-EQ_{r-1} is strictly stronger than UAV-EQ_r (for each constant r). The analogous result for UAV-ARB-EQ is valid. Furthermore, $\text{UAV-MQ}_{r+\omega(\log n)}$ is strictly stronger than UAV-MQ_r. We also determine the relation between query types chosen from different hierarchies.

Literatur

[1] Dana Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, April 1988.

[2] Sally A. Goldman, Stephen Kwek, and Stephen D. Scott. Learning from examples with unspecified attribute values. In *Proceedings of the 10th Annual Conference on Computational Learning Theory*, pages 231–242. ACM Press, New York, NY, 1997.

Attribut-effizientes Lernen (Abstract)

Peter Damaschke
FernUniversität, Theoretische Informatik II
58084 Hagen, Germany
Peter.Damaschke@fernuni-hagen.de

Wir untersuchen in [4] das Lernen von beliebigen Booleschen Funktionen, in denen nur r der n Variablen relevant sind, mittels Orakelfragen. Probleme dieses Typs haben interessante Anwendungen [1] [5] [7]. Die bisherige Literatur (etwa [2] [3] [8]) konzentriert sich auf spezielle Funktionenklassen und betrachtet nur adaptive Strategien (d.h. die Fragen dürfen von vorherigen Antworten abhängen). In manchen Anwendungen sind die Orakelfragen (Tests) jedoch zeitraubend, sind aber andererseits parallel ausführbar, so daß auch nichtadaptive Strategien gefragt sind.

Wir geben zunächst einen simplen, aber fast optimalen adaptiven Algorithmus mit $O(r2^r \log n)$ Fragen an. Es stellt sich heraus, daß alle bis auf $r \log n$ Fragen eine r-universelle Belegungsfamilie bilden müssen und nichtadaptiv gestellt werden können. Im Fall $r = 2$ kommen wir mit insgesamt etwa $2.275 \log n$ Fragen aus, was nahe an der offensichtlichen unteren Schranke $2 \log n$ liegt.

Es folgt eine kombinatorische Charakterisierung derjenigen Belegungsfamilien, welche die genannten Funktionen rein nichtadaptiv lernen – sog. r-weise bipartit zusammenhängende Familien. Es existieren solche Familien der Größe $O(r2^r \log n)$ – das bedeutet: Nichtadaptives attribut-effizientes Lernen ist nicht sehr viel teurer als adaptives Lernen. Eine pseudopolynomiale explizite Konstruktion mittels einer Derandomisierungs-Technik aus [6] hat eine kaum schlechtere Schranke.

Im Falle monotoner Boolescher Funktionen kann die optimale Anzahl von $O(2^r + r \log n)$ Fragen bereits in $O(r)$ parallelen Runden erreicht werden (unabhängig von n), rein nichtadaptives Lernen benötigt dagegen wieder $\Omega(2^r \log n)$ Fragen.

Literatur

[1] D.J. Balding, D.C.Torney: A comparative survey of non-adaptive pooling designs, in: Genetic Mapping and DNA Sequencing, *IMA Volumes in Mathematics and Its Applications*, Springer 1995, 133-155

[2] A.Blum, L.Hellerstein, N.Littlestone: Learning in the presence of finitely or infinitely many irrelevant attributes, *J. Comp. System. Sc.* 50 (1995), 32-40

[3] N.H.Bshouty, L.Hellerstein: Attribute-efficient learning in query and mistake-bound models, *9th COLT'96*, 235-243

[4] P.Damaschke: Adaptive vs. nonadaptive attribute-efficient learning, *30th ACM STOC* (1998), Dallas/Texas

[5] M.Farach, S.Kannan, E.Knill, S.Muthukrishnan: Group testing problems in experimental molecular biology, *Proc. of Sequences'97*

[6] M.Naor, L.J.Schulman, A.Srinivasan: Splitters and near-optimal derandomization, *36th IEEE FOCS* (1995), 182-191

Information Theory 34 (1988), 513-522

[8] R.Uehara, K.Tsuchida, I.Wegener: Optimal attribute-efficient learning of disjunction, parity, and threshold functions, *EuroCOLT'97*, *LNAI* 1208 (Springer), 171-184

Which Polynomial $\mathcal{D}$–Verboseness Classes Contain p-Bi-Immune Languages?

Arfst Nickelsen
Fachbereich Informatik der TU Berlin
nicke@cs.tu-berlin.de

Zusammenfassung

The connection is investigated between two well known notions which deal with languages that show polynomial time behaviour weaker than membership decidability. One notion is polynomial time bi-immunity (p-bi-immunity). The other one is polynomial time $\mathcal{D}$-verboseness which captures p-selectivity, p-cheatability, p-verboseness and similar notions, where partial information about the characteristic function is computed. The type of partial information is determined by a family of sets of bitstrings $\mathcal{D}$. A full characterization of those $\mathcal{D}$ for which there are p-bi-immune polynomially $\mathcal{D}$-verbose languages is given. Results of the same type for special cases of polynomial $\mathcal{D}$-verboseness were already given by Goldsmith, Joseph, Young [GJY93], Beigel [Bei90], and Amir, Gasarch [AG88].

1 Introduction

If a language is not decidable in polynomial time one may ask wether it nevertheless exhibits some polynomial time behaviour. One then does not expect the polynomial time algorithm for the language A to actually answer the question "$x \in A$?" for all inputs x. Instead one weakens this demand in different ways. One way of weakening is to expect the polynomial time algorithm to decide only an infinite subset of A or of $\overline{A}$, the complement of A. If even this is not possible, A is called polynomially bi-immune or p-bi-immune. Another approach is to run the algorithm on tuples of input words $(x_1, \ldots, x_n)$ and expect some partial information on membership of these words in A. This means that the algorithm should narrow the range of possibilities for values of $\chi_A(x_1, \ldots, x_n)$ (where χ_A is the characteristic function for A). To get a unified picture of such classes Beigel, Gasarch and Kinber introduced $\mathcal{D}$-verboseness and strong $\mathcal{D}$-verboseness [BGK95] where the type of partial information is specified by a family $\mathcal{D}$ of sets of bitstrings. The definitions for the time bounded version, namely polynomially $\mathcal{D}$-verbose languages, are given below. Basic properties of these polynomial $\mathcal{D}$-verboseness classes are presented in [Nic97]. The purpose of this paper is to fully characterize those $\mathcal{D}$ for which languages can be at the same time polynomially $\mathcal{D}$-verbose and p-bi-immune.

2 Definitions

For a given language A the characteristic function $\chi_A : \Sigma^* \to \{0, 1\}$ is defined by $\chi_A(x) = 1 \Leftrightarrow x \in A$. We extend χ_A to tuples of words by $\chi_A(x_1, \ldots, x_n) = \chi_A(x_1) \cdots \chi_A(x_n)$. For a bitstring b the number of 1's in b is denoted $\#_1(b)$, $b[i]$ is the i-th bit of b, and $b[i, \ldots, j]$ is the string formed by the i-th to the j-th bit of b. We now define p-bi-immunity and polynomial $\mathcal{D}$-verboseness.

Definition 1 (p-Bi-Immunity) *A language A is p-bi-immune if neither A nor $\overline{A}$ contains an infinite language $B \in P$.*

strings of length n, i.e., $D_i \subseteq \{0,1\}^n$ for all $i \in \{1,\dots,r\}$. We call $\mathcal{D}$ an n-family if $\bigcup_{i=1}^{r} D_i = \{0,1\}^n$.

Definition 3 (Polynomially $\mathcal{D}$-Verbose) *For a given n-family $\mathcal{D}$ a language A is in $P[\mathcal{D}]$ iff there is a polynomially time-bounded deterministic Turing machine M that on input $(x_1,\dots,x_n)$ outputs a $D \in \mathcal{D}$ such that $\chi_A(x_1,\dots,x_n) \in D$. Such languages are called polynomially $\mathcal{D}$-verbose.*

For more details on polynomial $\mathcal{D}$-verboseness see [Nic97]. We restate some known fact on n-families and $\mathcal{D}$-verboseness.

Definition 4 (Normal Form) *An n-family $\mathcal{D}$ is in normal form if it is closed under permutations, projections and replacements.*

Fact 5 (Normal Form) *For every n-family $\mathcal{D}$ there is a unique n-family $\mathcal{D}'$ in normal form with*

$$P[\mathcal{D}] = P[\mathcal{D}'] \ .$$

Fact 6 (Class Inclusion Reduces to Family Inclusion) *For n-families $\mathcal{D}_1$, $\mathcal{D}_2$ in normal form*

$$P[\mathcal{D}_1] \subseteq P[\mathcal{D}_2] \text{ iff } \mathcal{D}_1 \subseteq \mathcal{D}_2 \ .$$

Definition 7 (Generated n-Family) *Consider sets of bitstrings $D_1,\dots,D_r \subseteq \{0,1\}^n$. Then $\langle D_1,\dots,D_r\rangle$ denotes the minimal n-family $\mathcal{D}$ in normal form such that $\{D_1,\dots,D_r\} \subseteq \mathcal{D}$. This means that $\langle D_1,\dots,D_r\rangle$ is the closure of $\{D_1,\dots,D_r\}$ under permutations, projections and replacements. We say that $\langle D_1,\dots,D_r\rangle$ is generated by $D_1,\dots,D_r$.*

Some n-families are of special interest. We define n-SEL and (k,n)-SIZE.

Definition 8 (SEL, SIZE)

- *n-SEL $= \langle\{0^i 1^{n-i} \mid 0 \le i \le n\}\rangle$*
- *For $1 \le k \le 2^n$: (k,n)-SIZE $= \{D \subseteq \{0,1\}^* \mid |D| \le k\}$*

The class P [2-SEL] equals the class P-SEL of p-selective languages. We get:

Fact 9 (SEL)

$$P\text{-}SEL = P[2\text{-}SEL] = P[n\text{-}SEL] \text{ for } n \ge 2$$
$$P[(k,k)\text{-}SIZE] = P[(k,n)\text{-}SIZE] \text{ for } n \ge k$$

3 Previous Results

For polynomial $\mathcal{D}$-verboseness three results about the possibility of p-bi-immunity are known. We restate these results in our nomenclature. Goldsmith, Joseph and Young [GJY87][GJY93] showed the following.

Theorem 10 (Goldsmith, Joseph, Young)
$P[2\text{-}SEL]$ contains bi-immune languages.

In the same paper Goldsmith, Joseph and Young [GJY93] give a construction that yields the following result which was independently obtained by Beigle [Bei90].

Theorem 11 (Beigel; Goldsmith, Joseph, Young)
$P[\langle\{000, 001, 010, 011\}\rangle]$ contains a p-bi-immune language.

On the other hand Amir and Gasarch [AG88] showed:

Theorem 12 (Amir, Gasarch)
$P[(2,2)\text{-}SIZE]$ does not contain p-bi-immune languages.

We first define two special types of families that we need to state the main result.

Definition 13 (n-TOP, n-BOTTOM) *For $n \geq 2$ define:*

- $D^n_{top} = \{b \in \{0,1\}^n \mid \#_1(b) = n \text{ or } \#_1(b) = n-1\}$
- $D^n_{bottom} = \{b \in \{0,1\}^n \mid \#_1(b) = 0 \text{ or } \#_1(b) = 1\}$

These sets of bitstrings are used to define for every n two n-families:

- $n\text{-}TOP = \langle D^n_{top} \rangle$
- $n\text{-}BOTTOM = \langle D^n_{bottom} \rangle$

Now we can state our main result. It gives a characterization of the polynomial verboseness classes that contain p-bi-immune languages.

Theorem 14 (Main Theorem) *Let $\mathcal{D}$ be an n-family in normal form. Then*

$P[\mathcal{D}]$ does not contain p-bi-immune languages iff $\mathcal{D} \subseteq n\text{-}BOTTOM$ or $\mathcal{D} \subseteq n\text{-}TOP$.

We have to postpone the proof because we first introduce some new definitions and prove some lemmas that will be used in the proof of Theorem 14 given at the end of the section. Because of lack of space we can not give all the proofs. We restrict ourselves to the most dificult proof. The following lemma and Theorem 16 will be used to prove the *if*-part of the proof of the Main Theorem.

Lemma 15 *For all $n \geq 2$:*

$$P[n\text{-}BOTTOM] = P[2\text{-}BOTTOM] \text{ and } P[n\text{-}TOP] = P[2\text{-}TOP]$$

Theorem 16

$$A \in P[2\text{-}BOTTOM] \Rightarrow A \text{ is not p-bi-immune.}$$
$$A \in P[2\text{-}TOP] \Rightarrow A \text{ is not p-bi-immune.}$$

To prove the *only if*-part of the Main Theorem we look for the *minimal* families in normal form that are not subfamilies of n-BOTTOM or n-TOP.

Definition 17 *For $n \geq 2$ define three special sets of bitstrings D^n_s, D^n_t, and D^n_b as follows:*

- $D^n_s = \{000^{n-2}, 010^{n-2}, 110^{n-2}\}$
- $D^n_t = \{011^{n-2}, 101^{n-2}, 111^{n-2}\}$
- $D^n_b = \{000^{n-2}, 010^{n-2}, 100^{n-2}\}$

The indices s, t, and b are meant to remind of selectivity, top and bottom. We show that $\langle D^n_s \rangle$ and $\langle D^n_t, D^n_b \rangle$ are the minimal families we are looking for:

Theorem 18 *For every n-family $\mathcal{D}$ in normal form exactly one of the following cases holds:*

1. *$\mathcal{D} \subseteq n\text{-}TOP$ or $\mathcal{D} \subseteq n\text{-}BOTTOM$*
2. *$\langle D^n_s \rangle \subseteq \mathcal{D}$ or $\langle D^n_t, D^n_b \rangle \subseteq \mathcal{D}$*

Proof We construct a tally set $A \subseteq \{1\}^*$ that is in $\mathrm{P}[\langle D_t^n, D_b^n\rangle]$ and p-bi-immune. We will diagonalize against every Turing machine that could possibly decide an infinite subset of A or $\overline{A}$ (where $\overline{A}$ is $\{1\}^* \setminus A$). Let $\langle M_k\rangle_{k\in\mathbb{N}}$ be a standard enumeration of polynomial time Turing machines such that the running time of M_k is bounded by a polynomial p_k, say $n^{\log k} + \log k$. We have to ensure that for every M_k that accepts infinitely many words there are words $w_1, w_2 \in L(M_k)$ with $w_1 \in A$ and $w_2 \in \overline{A}$. For every $n \in \mathbb{N}$ we define stepwise approximations to A and to $\overline{A}$. For this purpose we use a sequence of natural numbers $\langle n_k\rangle_{k\in\mathbb{N}}$ that grows fast enough that on inputs of length $\geq n_i$ we can simulate all M_j with $j \leq i-1$ on all words of length $\leq n_{i-1}$. It is sufficient to define $n_0 = 1$, $n_{i+1} = 2^{n_i}$. During the construction of A and $\overline{A}$ we keep track of a bunch of parameters. After step n we will have constructed four disjoint sets IN_n, OUT_n, OLD_n and NEW_n such that $\mathrm{IN}_n \cup \mathrm{OUT}_n \cup \mathrm{OLD}_n \cup \mathrm{NEW}_n = \{1\}^{\leq n}$. For all n it will hold that $\mathrm{IN}_n \subseteq \mathrm{IN}_{n+1}$ and $\mathrm{OUT}_n \subseteq \mathrm{OUT}_{n+1}$. In the end we define $A = \bigcup_n \mathrm{IN}_n$ and the construction will then yield $\overline{A} = \bigcup_n \mathrm{OUT}_n$.
The sets OLD_n and NEW_n contain those words up to length n for which membership in A or $\overline{A}$ is not decided after step n. We also maintain a finite list $L_n \subseteq \mathbb{N}$ of indices of Turing machines. An index k enters the list at construction step n_k and is removed from the list at a later step n only if both requirements for M_k are fulfilled, i.e. there are $w_1 \in \mathrm{IN}_n$ and $w_2 \in \mathrm{OUT}_n$ with $M_k(w_1) = \text{accept}$ and $M_k(w_2) = \text{accept}$.
Which requirements are still unfulfilled after step n is expressed by the function r_n that maps every k to a subset of $\{in, out\}$. E.g., $r_n(k) = \{in\}$ means that it is still required to put a w_1 with $M_k(w_1) = \text{accept}$ into A at some later stage, but there already is a $w_2 \in \mathrm{OUT}_n$ with $M_k(w_1) = \text{accept}$. We also need a parameter status_n with possible values t and b. If $\mathrm{status}_n = b$ then at most one of the sets OLD_n and NEW_n will become part of A in a later step, if $\mathrm{status}_n = t$ then at least one of the sets OLD_n and NEW_n will enter A later on.
As a last parameter we need $\mathrm{index}_n \in L_n \cup \{\infty\}$. If after a construction step $\mathrm{index}_n = k \in L_n$ this means that there is a requirement for M_k that we would like to fulfill by putting a $w \in \mathrm{OLD}_{n-1}$ into A or $\overline{A}$ but we are at the moment hindered to do this by the current value of status_n. Therefore we have to wait to do so until the status has changed or until the requirement is overruled by a requirement with higher priority, i.e. by a requirement for a machine $M_{k'}$ with $k' < k$.
Now let IN_{n-1}, OUT_{n-1}, OLD_{n-1}, NEW_{n-1}, L_{n-1}, r_{n-1}, status_{n-1} and index_{n-1} be already constructed. At step n consider the new word 1^n. Check wether $n = n_i$ for some i.
Case 1: $n \neq n_i$ for all i. Add the new word 1^n to the set NEW, leave everything else unchanged.

$$\mathrm{IN}_n = \mathrm{IN}_{n-1} \text{ and } \mathrm{OUT}_n = \mathrm{OUT}_{n-1}$$

$$\mathrm{OLD}_n = \mathrm{OLD}_{n-1} \text{ and } \mathrm{NEW}_n = \mathrm{NEW}_{n-1} \cup \{1^n\}$$

$$L_n = L_{n-1} \cup \{i\} \text{ and } r_n = r_{n-1}$$

$$\mathrm{status}_n = \mathrm{status}_{n-1} \text{ and } \mathrm{index}_n = \mathrm{index}_{n-1}$$

Case 2: $n = n_i$ for some i. Suppose that $\mathrm{status}_{n-1} = b$. (In case $\mathrm{status}_{n-1} = t$ we have analogous subcases. See explanation below.) For every $k \in L_{n-1}$ and for every $x \in \mathrm{OLD}_{n-1}$ compute $M_k(x)$. Three different cases can occur:
Case 2.1: $M_k(x) = \text{reject}$ for all k and x. No requirement for $k < \mathrm{index}_{n-1}$ can be fulfilled at this stage. On the other hand there is nothing wrong in putting OLD_{n-1} into $\overline{A}$. Therefore we add OLD_{n-1} to OUT_{n-1}, change the status from b to t and give up the restriction on indices possibly imposed by index_{n-1}.

$$\mathrm{IN}_n = \mathrm{IN}_{n-1} \text{ and } \mathrm{OUT}_n = \mathrm{OUT}_{n-1} \cup \mathrm{OLD}_{n-1}$$

$$\mathrm{OLD}_n = \mathrm{NEW}_{n-1} \text{ and } \mathrm{NEW}_n = \{1^n\}$$

Case 2.2: $M_k(x) =$ accept for some k and x. Choose k_0 as the minimal k for which this happens and let x_0 be the smallest word in OLD_{n-1} with $M_{k_0}(x_0) =$ accept. We distinguish two subcases depending on the value of $r_{n-1}(k_0)$.
Case 2.2.1: $out \in r_{n-1}(k_0)$. We can directly fulfill the requirement for k_0 by putting x_0 (and the whole set OLD_{n-1}) into $\overline{A}$. If both requirements for k_0 are then fullfilled we remove k_0 from the list of machine indices; else we remove the requirement *out* from $r_{n-1}(k_0)$. We also change the status to t.

$$\mathrm{IN}_n = \mathrm{IN}_{n-1} \text{ and } \mathrm{OUT}_n = \mathrm{OUT}_{n-1} \cup \mathrm{OLD}_{n-1}$$

$$\mathrm{OLD}_n = \mathrm{NEW}_{n-1} \text{ and } \mathrm{NEW}_n = \{1^n\}$$

$$\text{if } r_{n-1}(k_0) = \{in, out\} \text{ then } L_n = L_{n-1} \cup \{i\}$$

$$\text{if } r_{n-1}(k_0) = \{out\} \text{ then } L_n = (L_{n-1} \setminus \{k_0\}) \cup \{i\}$$

$$r_n(k_0) = r_{n-1}(k_0) \setminus \{out\} \text{ and } r_n(k) = r_{n-1}(k) \text{ for } k \neq k_0$$

$$\mathrm{status}_n = t \text{ and } \mathrm{index}_n = \infty$$

Case 2.2.2: $r_{n-1}(k_0) = \{in\}$ We would like to put OLD_{n-1} including x_0 into A, but we can't do this at this moment because then we would have to put NEW_{n-1} into $\overline{A}$ (remember that in status b at most one of the sets OLD_{n-1} and NEW_{n-1} may enter A). But maybe we need words from NEW_{n-1} to fulfill *in*-requirements with higher priority. Therefore we only change index to k_0 without fulfilling any requirement. We let the sets OLD and NEW change their roles.

$$\mathrm{IN}_n = \mathrm{IN}_{n-1} \text{ and } \mathrm{OUT}_n = \mathrm{OUT}_{n-1}$$

$$\mathrm{OLD}_n = \mathrm{NEW}_{n-1} \text{ and } \mathrm{NEW}_n = \mathrm{OLD}_{n-1} \cup \{1^n\}$$

$$L_n = L_{n-1} \cup \{i\},\ r_n = r_{n-1},\ \mathrm{status}_n = \mathrm{status}_{n-1} \text{ and } \mathrm{index}_n = k_0$$

How do we deal with Case 2 if $\mathrm{status}_{n-1} = t$? Mainly the roles of IN and OUT are exchanged. Details are omitted. This ends the description of the procedure. We now state some properties of the construction.

Claim 1 Suppose that after some stage n $x \in \mathrm{OLD}_n$ and $y \in \mathrm{NEW}_n$. If $\mathrm{status}_n = t$, then $x \in A$ or $y \in A$. If $\mathrm{status}_n = b$, then $x \notin A$ or $y \notin A$.

This claim is verified by considering the different cases that can happen during the procedure.

Claim 2 For every n there is a step m in the construction where 1^n enters IN_m or OUT_m.

The word 1^n enters NEW at step n. Suppose $n_{i-1} \leq n < n_i$. Then at step n_i 1^n enters OLD. At step n_{i+1} one of the Cases 2.1, 2.2.1 or 2.2.2 occurs. In Case 2.1 and Case 2.2.1 1^n is moved to IN or OUT. Suppose Case 2.2.2 occurs and index is set to k_0. Then together with 1^n there is a word x_0 in OLD with $M_{k_0}(x_0) =$ accept. Now 1^n and x_0 can possibly oscillate from OLD to NEW and back again. But each time 1^n and x_0 are in NEW it holds that index $< k_0$. Therefore the open requirement for k_0 can not be fulfilled in a situation where $1^n \in$ NEW.
How many oscillations can happen without putting 1^n into IN or OUT? If at a step n_j 1^n is in OLD_{n_j-1} either

- 1^n is put into IN or OUT or
- index is changed from ∞ to a $k \leq k_0$ or

- a requirement is fulfilled for a $k < k_0$.

Therefore at most $k_0 \cdot 2(k_0 - 1)$ oscillations can happen before 1^n moves to IN or OUT.

Claim 3 For each k where $L(M_k)$ is infinite at some step of the procedure $r(k)$ is empty.

Assume that k_0 is the smallest k where $L(M_k)$ is infinite and $r_n(k) \neq \emptyset$ for all n. Let m_0 be such that

- for all $k < k_0$ with $L(M_k)$ infinite $r_{m_0} = \emptyset$ and
- for all $k < k_0$ with $L(M_k)$ finite is $L(M_k) \subseteq \{1\}^{m_0 - 1}$ and
- $m_0 > n_{k_0}$.

Consider $m_1 = \min\{m \mid 1^m \in L(M_k), m > m_0\}$. Assume w.l.o.g. that $r_{m_1} = \{in\}$. If $n_{i-1} \leq m_1 < n_i$ then in step n_i 1^{m_1} enters OLD, k_0 is in the requirement list L and there is no requirement left with higher priority. This means that in step n_{i+1} either Case 2.2.1 or 2.2.2 occurs. In Case 2.2.1 where status $= t$ the in-requirement is fulfilled which contradicts the assumption. In Case 2.2.2 where status $= b$ index is set to k_0. Then at step n_{i+2} Case 2.1 occurs (because there are no k in list L with $k <$ index). The status is changed to t, 1^{m_1} is in OLD again and at step n_{i+3} 1^{m_1} will be moved to IN because Case 2.2.1 occurs.
Claim 2 implies that $\overline{A} = \bigcup_n \mathrm{OUT}_n$. Claim 3 implies that A indeed is p-bi-immune.

Claim 4 The construction of IN_n, OUT_n, OLD_n, NEW_n, and status_n can be done in time polynomial in n.

This is the case because we have only to simulate the machines M_k for $k \in L_{n-1}$ on words of length n_{i-1} when $n_i \leq n$.

Claim 5 A is in $\mathrm{P}[\langle D_t^n, D_b^n \rangle]$.

Consider an input $(x_1, \ldots, x_n)$. If $m = \max_i |x_i|$ compute IN_n, OUT_n, OLD_n, NEW_n, and status_n. Claim 4 ensures that this can be done in polynomial time.
Suppose $\mathrm{status}_m = b$ (the case $\mathrm{status}_m = t$ is treated analogously). Suppose that there are i and j, $i < j$ with $x_i \in \mathrm{OLD}_m$ and $x_j \in \mathrm{NEW}_m$ (or $x_i \in \mathrm{NEW}_m$ and $x_j \in \mathrm{OLD}_m$). Because of Claim 1 we know that $\chi_A(x_1, \ldots, x_n)[i,j] \in \{00, 01, 10\}$. For x_l with $l \neq i, j$ different cases can occur.

- If $x_l \in \mathrm{IN}_m$ then $\chi_A(x_l) = 1$.
- If $x_l \in \mathrm{OUT}_m$ then $\chi_A(x_l) = 0$.
- If x_l and x_i are both in OLD_m (or both in NEW_m) then $\chi_A(x_l) = \chi_A(x_i)$.
- If x_l and x_j are both in NEW_m (or both in OLD_m then $\chi_A(x_l) = \chi_A(x_j)$.

Therefore we can apply projections π_l^1 or π_l^0 or replacements $\rho_{i,l}$ or $\rho_{j,l}$ to determine the bits of $\chi_A(x_1, \ldots, x_n)[l]$ depending on $\chi_A(x_1, \ldots, x_n)[i,j]$. Thus we get a set D of three bitstrings containing $\chi_A(x_1, \ldots, x_n)$ and $D \in \langle D_b^n \rangle$. $\square$

Lemma 20 *For all $n \geq 3$ $P[\langle D_s^n \rangle] = P[\langle D_s^3 \rangle]$.*

Theorem 21 *For all $n \geq 2$ the class $P[\langle D_s^3 \rangle]$ contains a p-bi-immune language A.*

Proof of Main Theorem
if-part: We want to show that if $\mathcal{D} \subseteq n$-TOP or $\mathcal{D} \subseteq n$-BOTTOM then P $[\mathcal{D}]$ does not contain p-bi-immune languages. It suffices to show that P $[n$-TOP$]$ and P $[n$-BOTTOM$]$ do not contain p-bi-immune languages. Because of Lemma 15 it suffices to show that P [2-TOP] and P [2-BOTTOM] do not contain p-bi-immune languages. But this holds by Theorem 16.
only if-part: For this direction we want to show that for $\mathcal{D}$ with $\mathcal{D} \not\subseteq n$-TOP and $\mathcal{D} \not\subseteq n$-BOTTOM P $[\mathcal{D}]$ contains p-bi-immune languages. By Theorem 18 we know that $\langle D_s^n \rangle \subseteq \mathcal{D}$ or $\langle D_t^n, D_b^n \rangle \subseteq \mathcal{D}$. Therefore it suffices to show that P $[\langle D_s^n \rangle]$ and P $[\langle D_t^n, D_b^n \rangle]$ contain p-bi-immune languages. By Theorem 19 P $[\langle D_t^n, D_b^n \rangle]$ contains p-bi-immune languages. By Lemma 20 we know P $[\langle D_s^n \rangle] =$ P $[\langle D_s^3 \rangle]$. By Theorem 21 P $[\langle D_s^3 \rangle]$ contains p-bi-immune languages. This finishes the proof. □
If we specialize the Main Theorem to tuple-length two and to SIZE-classes we get the following corrolaries:

Corollary 22 (Tuple-Length 2)

- *P* $[(2,2)$-*CARD*$]$ *contains p-bi-immune languages.*
- *P* $[2$-*TOP*$]$ *does not contain p-bi-immune languages.*
- *P* $[2$-*BOTTOM*$]$ *does not contain p-bi-immune languages.*

Corollary 23 (SIZE-Classes)
For $n \geq 2$

- *P* $[(k,n)$-*SIZE*$]$ *contains p-bi-immune languages iff* $k \geq 3$.

Literatur

[AG88] A. Amir and W. Gasarch. Polynomial terse sets. *Information and Computation*, 77, 1988.

[Bei90] R. Beigel. Bi-immunity results for cheatable sets. *TCS*, 73(3), 1990.

[BGK95] R. Beigel, W. Gasarch, and E. Kinber. Frequency computation and bounded queries. In *Proc. 10th Structure in Complexity Theory*, 1995.

[BKS92] R. Beigel, M. Kummer, and F. Stephan. Quantifying the amount of verboseness. In *Proc. Logical Found. of Comp. Sc.* LNCS 620, 1992.

[BKS95] R. Beigel, M. Kummer, and F. Stephan. Approximable sets. *Information and Computation*, 120(2), 1995.

[GJY87] J. Goldsmith, D. Joseph, and P. Young. Self-reducible, P-selective, near-testable, and P-cheatable sets: The effect of internal structure on the complexity of a set. In *Proceedings 2nd Structure in Complexity Theory Conference*, pages 50–59. IEEE Computer Society Press, 1987.

[GJY93] J. Goldsmith, D. Joseph, and P. Young. A note on bi-immunity and p-closeness of p-cheatable sets in P/Poly. *JCSS*, 46(3), 1993.

[Nic97] A. Nickelsen. On polynomially $\mathcal{D}$-verbose sets. In *Proc. STACS 97*, pages 307–318, 1997.

Protocols – Methods for the Verification of Data Link Protocols

Christoph Meinel, Christian Stangier[1]
FB IV - Informatik, Universität Trier
D-54286 Trier, Germany
email: {meinel,stangier}@uni-trier.de

The demand for formal verification of communication protocols is increasing since the use of distributed systems is still rapidly growing. Ordered binary decision diagrams (OBDDs) are widely and successfully used in the area of digital circuit verification. This suggests an application of OBDD techniques to the verification of protocols, but the formal modelling of a complete protocol is too complex for a verification with OBDDs using current techniques. Therefore, the model has to be restricted. The model we are using is as least restricted as needed and preserves as much of the protocol's properties as possible. Since there is no further knowledge about protocol verification using OBDD techniques, we decided to use two common data link protocols to gain experience. The experimental results are leading to an approach to the problem of finding well suited orders of input variables needed for an efficient OBDD representation. Furthermore, we introduce a technique for avoiding time consuming computations by using additional hardware. As a result, we have obtained general knowledge on OBDD-based communication protocol verification that will be applied to more complex protocols.

[1]Supported by the DFG-Graduiertenkolleg "Mathematische Optimierung"

Translating Equality Downwards

Edith Hemaspaandra[†]
Dep. of Computer Science
Le Moyne College
Syracuse, NY 13214, USA

Lane A. Hemaspaandra[‡]
Dep. of Computer Science
University of Rochester
Rochester, NY 14627, USA

Harald Hempel[§]
Inst. für Informatik
Friedrich-Schiller-Universität Jena
07743 Jena, Germany

Abstract

Downward translation of equality refers to cases where a collapse of some pair of complexity classes would induce a collapse of some other pair of complexity classes that (a priori) one expects are smaller. Recently, the first downward translation of equality was obtained that applied to the polynomial hierarchy—in particular, to bounded access to its levels [HHH]. In this paper, we provide a much broader downward translation that extends not only that downward translation but also that translation's elegant enhancement by Buhrman and Fortnow [BF96]. Our work also sheds light on previous research on the structure of refined polynomial hierarchies [Sel95, Sel94], and strengthens the connection between the collapse of bounded query hierarchies and the collapse of the polynomial hierarchy.

Literatur

[BF96] H. Buhrman and L. Fortnow. Two queries. Technical Report 96-20, University of Chicago, Department of Computer Science, Chicago, IL, September 1996.

[HHH] E. Hemaspaandra, L. Hemaspaandra, and H. Hempel. A downward collapse within the polynomial hierarchy. *SIAM Journal on Computing*. To appear.

[Sel94] V.L. Selivanov. Two refinements of the polynomial hierarchy. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science*, pages 439–448. Springer-Verlag *Lecture Notes in Computer Science #775*, February 1994.

[Sel95] V.L. Selivanov. Fine hierarchies and boolean terms. *Journal of Symbolic Logic*, 60(1):289–317, 1995.

[†]Email: `edith@bamboo.lemoyne.edu`. Supported in part by grant NSF-INT-9513368/DAAD-315-PRO-fo-ab. Work done in part while visiting Friedrich-Schiller-Universität Jena.

[‡]Email: `lane@cs.rochester.edu`. Supported in part by grants NSF-CCR-9322513 and NSF-INT-9513368/DAAD-315-PRO-fo-ab. Work done in part while visiting Friedrich-Schiller-Universität Jena.

[§]Email: `hempel@informatik.uni-jena.de`. Supported in part by grant NSF-INT-9513368/DAAD-315-PRO-fo-ab.

Learning Dead Ends in Sokoban

Stefan Edelkamp, Stefan Schrödl [*]

Zusammenfassung

A major issue in heuristic search is the detection of *dead ends*, i.e., positions which cannot possibly be solved on any available branch. We present a general algorithm ADP^* as a conservative extension of the well-known A^*-algorithm to deal with such cases. It detects and generalizes dead end patterns; they are stored and utilized to prune the search space.

Recently, the *Sokoban* puzzle was considered to be a challenge for AI search techniques. We chose this domain for evaluating our algorithm, and present experimental results.

1 Introduction

The main goal in Heuristic Search is to control huge (in general exponential sized) search spaces. Two different orthogonal approaches can be distinguished: First, to devise an *heuristic* that estimates the optimal solution length for every state in the space as accurate as possible. Second, in several domains there are one-way moves that can never be went back on (doors may shut if we go through, or handles might even be installed only on one side). *Pruning* is used to discard such *dead ends*, i.e., branches that cannot possibly lead to a goal.

In this paper we propose an extension of the well known A^* algorithm proposed by Hart, Nilsson and Raphael (1968) that allows to detect, memorize and generalize situations that are dead. Herein, generalization is done by restricting to relevant (partial) subpositions responsible for the failure. A data structure called *Subposition Store* is used for efficient storage and retrieval of these subposition. The idea of generalisation is closely related to duplicate pruning such as in Edelkamp (1997).

The algorithm is evaluated in the domain of the *Sokoban* puzzle, which is one of the remaining one-person games in which the human solution quality still outperforms all attempts to automatic solving strategies coded in a computer program. In Sokoban n balls are placed somewhere in a maze containing n goal fields which they must eventually reach. The player controls a man which can traverse the board and push the balls onto adjacent empty squares. Three problems can be distinguished: *Decide*, *Pushes* and *Moves*. *Decide* is just the task to solve the puzzle. *Pushes* additionally asks to minimize the number of ball pushes whereas *Moves* request an optimal number of man movements. Although all problems are computational equivalent the actual search spaces differ. Sokoban is proven to be *NP*-hard by Culbersone (1997) for a growing board and number of balls but polynomial for a fixed number of balls. A solution for *Moves* and *Pushes* implies a solution for *Decide*, but optimal solutions to *Moves* and *Pushes* fail to imply each other.

Comparing man and machine can be done by studying the the set of 90 problems for the Sokoban Puzzle provided at http://xsokoban.lcs.mit.edu/xsokoban.html. Note that even when minimizing the number of moves the problem space has to be compressed to a weighted graph with each edge corresponding to a ball push. The weight of the edge is given by the shortest path from the current man position to the next ball to move.

[*]Institut für Informatik, Albert-Ludwigs-Universität, Am Flughafen 17, D-79110 Freiburg eMail: {edelkamp,schroedl}@informatik.uni-freiburg.de, Stefan Edelkamp is supported by DFG within graduate program on human and machine intelligence

of the one ball problems between each ball and each goal field. To calculate the matching we use an $O(n^3)$ minimum cost network flow algorithm (see Mehlhorn (1984)) based on n invocations of Dijkstra's original single source shortest path algorithm, which runs in $O(n^2)$. Furthermore, the heuristic can be refined, i.g. by counting the number of balls that are too closed together to enable the choice of the shortest path.

The underlying grid itself can be shorten to a weighted graph by introducing precomputed macro moves. We distinguish *tunnel macros* and *goal macros*. The former forwards a ball throughout a narrow corridor and the latter partitions the search space by directly placing balls onto empty goal fields when entering a goal room on an articulation square (i.g. a sole entrance of width one).

1.1 Detecting Immediate Dead Ends

We implemented the following procedure *stuck* to detect simple positions of this type. Call a ball *free* if it can possibly be pushed in any direction; that is, if it has two opposite adjacent empty squares either in horizontal or vertical direction. As it is removed, other balls may become mobile, in turn. We repeat this procedure and eventually, the status of all balls settles. If there are remaining balls the position is a dead-end: In order to free a ball with the man it is necessary that at least one of its neighbors has to be free.

The algorithm is implemented using a queue Q comprising all balls whose status could still change. Initially all balls are enqueued. Until Q gets empty, one element at time is examined; in the case that a dequeued unfree ball becomes free its unfree neighbors are inserted. The run time of the *stuck* algorithm is $O(n)$, since each ball that is found free only gives rise to a maximum of four neighbors to be enqueued.

1.2 Bottom-Up-Propagation

Procedure *stuck* described above can only detect a fraction of all the dead ends actually occurring. Let us now turn to additional methods in order to identify more of them. A position is dead if all successor positions are dead. In some leaf nodes in the search tree *stuck* determines the set of responsible balls: a dead subposition. To compute the dead subposition of the parent we add each ball which is already included in the dead subposition of at least one of the successors, and account for the respective moved ball.

1.3 Decomposition

A Sokoban position is dead if it contains a subset of balls that cannot be solved even if the remaining balls are removed. For example, this subposition could consist of the balls in an isolated, separate room. Thus, we can use *decomposition* to detect deadlocks more quickly by guessing subpositions and determining their solvability. Intuitively, these subpositions are "clusters" containing balls which are highly dependent on each other (e.g., blocking each other), and independent of the rest. We experimented with two different decomposition heuristics: 1) **cc:** Considering unreachable fields the following observation is central for our approach in finding dead positions: A position with a non-goal field on which the man can never get to is dead. Therefore, the first idea of decomposing a position is to take the graph G of all empty squares and partition G into connected rachable components using $O(B)$ time, with B being the size of the graph G. Collect all balls that are adjacent to one component and merge components that have a ball in common. If every empty square can be reached by the man the position is likely to be solvable. 2) **one:** This heuristic breaks an n-ball problem into an 1-ball problem and an $(n-1)$-ball problem by trying to push a single ball onto a goal field with the other balls congealed to walls.

therefore a shallow search. A good trade-off has to be found: the characteristics responsible for the dead end on the one hand should appear in only one component and, on the other hand, the problem parts should be far more easy to analyze than the original one.

2 The Subposition Store

Since dead end testing is done at each expansion, a fast implementation is crucial to the search algorithm. In Sokoban a pattern consists of one bit per square, indicating whether a ball resides on it. Therefore, we have to solve a *multiple two dimensional dictionary pattern matching problem.* One possibility would be an incremental solution. We could use a dictionary mapping each square to the patterns it is present in. Moreover, each position is equipped with an array of counters, one for each pattern keeping track of the number of matches. Since only two squares are affected per move, such a solution would have $O(p)$ time in the worst case, with p being the number of patterns. The add and delete lists of pattern according to each square are usually smaller than p in practice. Let L be the total number of balls of all pattern stored. Assuming a uniform distribution of the pattern on the board with board size B, the average run time turns out to be $O(L/B)$.

In depth first search (such as IDA^*, Korf (1985)), this would be feasible. However, in A^* the vector of p counters is to large to be retrieved at each expansion, so we chose a different approach that does not store information along with states. The straightforward solution would be to store each pattern as a bit vector of the board size B, and compare all patterns successively. By a suitable implementation using logical operations on a machine with word length w, this leads to an $O(pB/w)$ algorithm.

However, on the average, the number of balls n is considerably smaller than B. Our data structure *SubpositionStore (SPS)* allows to skip over squares that do not belong to any of the stored patterns. There are bit vectors associated with the squares of the board. The i-th pattern is stored in a distributed way among these vectors, namely in bit i of each one of them. When the number of patterns exceeds the word width, multiple words have to be provided.

Comparing the pattern store with a given position can proceed square by square. However, when the entry for some square consists entirely of zeros, we can skip to the next one. Thus, we can avoid a great deal of useless comparisons by jumping only from one significant square to another, using a linked list *next*. This takes $O(L)$ time in the worst case. The bitvector implementation can exploit the overlap of patterns to reduce execution to $O(L/w)$ in the best case.

Since testing can be finished as soon as all stored patterns have been disproved, further improvements can be achieved by appropriately arranging the order of squares. One possibility is sorting the *next* array according to the number of ones: it is reasonable to test those squares first that occur in most patterns. Even a decision tree can be built, depending on the outcome of the previous test. All these schemes exploit the overlap of patterns.

3 Algorithm ADP^*

Decomposition and bottom-up propagation can complement each other in a powerful way: the former one allows to find dead-end subpositions, and the latter one is able to combine these results to high-order structures.

Now we are ready to present our main algorithm ADP^* (for A^* with decomposition and propagation). It is a conservative extension of the well-known A^* algorithm, which consists of the underlined sections in Fig. 1. As usual, a priority queue PQ stores the set of open (horizon) nodes explored in the search tree; it is ordered according to the merit $f = g + h$, where g is the path length traveled so far since the root, and h an estimate of the remaining path to a goal. The hash of clarity, we abstract from the actual implementation by writing these data structures as sets.

*ADP** carries out expansion and decomposition in parallel, sharing the same priority queue PQ and hash table H. Solving the decomposed subpositions is inherently preferred to top level expansion, since the heuristic h estimating the goal distance tends to be lower for them. The top level of the search tree, generated exclusively by expansion, is explored in order to find the actual solution. Nodes generated by decomposition or which have any ancestor generated by decomposition are indicated in the algorithm by assigning them a *decompose*-flag. The sole purpose of these nodes is to determine solvability. Thus, if they are recognized as either dead or alive, they do not have to be further considered (expanded or decomposed). There might exist methods to determine that such a node can be solved, without explicitly computing a solution. Such a procedure analogous to the *stuck* function is referred to as *solvable* in Fig. 1. Note that this procedure may be an optimistic heuristic; more precisely, we might allow a small one-sided error, considering dead positions as alive, without affecting the correctness and optimality of the overall solution. To avoid recomputation of the status, we employ an additional hash table A. If a solvable node has been generated by expansion, then its parent must be alive, too; *BuPropAlive* propagates this status until a decomposition node is reached.

```
procedure $ADP^*$
  $\underline{PQ \leftarrow PQ \cup \{\text{root}\};\ H \leftarrow H \cup \{\text{root}\};}$
  $SPS \leftarrow \emptyset;$
  $\underline{\textbf{while } PQ \neq \emptyset \textbf{ do}}$
    $\underline{u \leftarrow min\{g(v)+h(v) \mid v \in PQ\};\ PQ \leftarrow PQ \setminus \{u\};}$
    $\underline{\textbf{if } (goal(u))}$
      if $(decompose(u))$ $BuPropAlive(u)$
      else $\underline{\textbf{return } path(u)}$
    elsif (($v \in SPS$ for some dead subposition $v$ of $u$) or $stuck(u)$)
      $BuPropDead(u)$
      if not ($decompose(u)$ and $u \in A$)
        if ($decompose(u)$ and $solvable(u)$)
          $BuPropAlive(u)$
        else
          $\underline{\Gamma(u) \leftarrow Expand(u)}$
          $\Delta(u) \leftarrow Decompose(u)$
          $\underline{\textbf{foreach } v \in \Gamma(u) \cup \Delta(u)}$
            $\underline{PQ \leftarrow PQ \cup \{v\};}$
            $\underline{\textbf{if } (v \in H)}$
              $\underline{g(v) \leftarrow min\{g(v), g(u)+1\}}$
            $\underline{\textbf{else}}$
              $\underline{g(v) \leftarrow g(u)+1;\ H \leftarrow H \cup \{v\};}$
```

Abbildung 1: The decomposition and bottom-up-propagation algorithm.

Until the priority queue gets empty, the node with lowest f value in PQ is chosen and removed. If it is a goal node and we are in the top level search we have found the optimal solution to the overall problem and are done, according to the correctness of the A^* algorithm that constitutes the underlined part in Fig. 1. After the goal-check we examine the status of the node by invoking the simple dead end detection algorithm *stuck*, the lookup procedure in the *Subposition Store* and the *solvable* procedure. If a node is found dead or alive we can propagate the news bottom-up

level	depth	exp	# p	exp	# p	exp
1	97	45	141	45	134	44
2	131	4144	220	2764	127	2485
3	134	468	204	167	143	252
4	355	>691320	1001	668522	241	> 291745
5	143	>822955	65	413541	55	>370385
6	110	1807	161	69	23	88
7	88	231752	125	159589	156	156370
17	213	>1068770	841	738254	255	>1059749
38	81	826069	331	38178	161	83929
78	136	159	14	159	9	159
80	231	3498	91	706	815	2996
82	135	>93969	122	563152	81	>93218
83	194	271	124	150	22	259

Tabelle 1: The effect of the learning dead end pattern.

and turn to the next one in the priority queue. Otherwise, the position remains undefined; it is decomposed into the set $\Delta(u)$ of subcomponents and expanded into the set $\Gamma(u)$ of successors found by performing all state transitions available. The ordinary successors are inserted, dropped and reopened as in usual A^*; for duplicates found on different paths, this includes changing the g value to the minimum of the two.

Depending on the given resources, decomposition can be either invoked in every step, once in a while or only in critical situation. *ADP** allows on-line or incremental learning: each dead subproblem found and inserted into the *Subposition Store* can be used immediately to prune the search tree and, therefore, to get deeper into the search tree. Some authors also refer to this aspect as *bootstrapping*.

4 Experimental Results

We test the *ADP** in the *Pushes* variant of the Sokoban puzzle. Up to now we have solved 13 levels of the bechmark xsokoban puzzle. We newly solved level 5, which Junghanns and Schaeffer (1997) could not handle. Each of them is executed five times. *ADP** is called twice, for decomposition according to heuristics of cc) and *one*. The resulting *Subposition Store* in both cases are used afterwards in two further straight searches without decomposition. The outcome is compared with a usual A^* algorithm. If the memory (2 million hashed states) is exhausted the search is abandoned.

Table 4 shows the number of balls, the optimal solution length, the number of expanded nodes for the different search schemes, and the total of learned dead end patterns in the *Subposition Store*.

The space complexity for our pruning approach with less than a thousand pattern is extremly small, compared to Junghanns and Schaeffer's 4.6 MByte dead end pattern database. In general, the decomposition *cc* leads to more and smaller patterns due the finer granularity of the partition; the smaller the pattern the higher the potential for generalisation. Therefore, the reduction in expansions is more pronounced than with *one*. Through the decomposition process it quite different the pruning effect of both strategies is not homogenous and might be used to complement each other.

A major issue in heuristic search is the detection of dead ends, i.e., positions which cannot possibly be solved on any available branch. In this paper we presented a general approach to deal with such cases. Our algorithm *ADP** detects and generalizes dead end patterns; they are stored and utilized to reduce the search space.

ADP* can be viewed as a pruning strategy which neglects branches that cannot lead to the goal. It is also possible to prune the search tree at branches that are guaranteed to have abbreviations or so-called *shortcuts*. Consider possible moves as a finite alphabet (e.g., U, D, L, R for a sliding tile puzzle), and branches as strings over it. Then a finite state machine can run in parallel to the search in order to predict that a given state has already been or will be visited on a shorter generating path. Such an automaton can be learned prior to the search (Taylor (1993)) or within the search (Edelkamp (1997)). Storing and retrieving the information for each state can be done in constant space and time. An important theoretical question for our algorithm is to design a *Subposition Store* that achieves the same complexity bound.

Literatur

[1] J. Culberson, *Sokoban is PSPACE-complete*, Technical Report TR-97-02, Departement of Computing Science, University of Alberta, 1997.

[2] S. Edelkamp, *Suffix Tree Automata in State Space Search*, KI'97: Lecture Notes in Artificial Intelligence (1303), 1997, pp 381–385.

[3] P. E. Hart and N. J. Nilsson and B. Raphael, *A Formal Basis for Heuristic Determination of Minimum Path Cost*, IEEE Trans. on SSC, 1968(4), p 100.

[4] A. Junghanns and J. Schaeffer, *Sokoban: A Challenging Single Agent Search Problem*, Workshop on Using Games as an Experimental Testbed for AI Research, IJCAI, 1997.

[5] R. E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, *Artificial Intelligence*, Vol. 27, No. 1, 1985, pp. 97-109.

[6] K. Mehlhorn, *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness* Springer Verlag, 1984.

[7] L. A. Taylor and R. E. Korf, *Pruning Duplicate Nodes in Depth-First Search*, Proceedings of the 11th National Conference on Artificial Intelligence, 1993, p. 756–761.

Christian Schindelhauer
Andreas Jakoby

Med. Uni. Lübeck, Institut für Theoretische Informatik,
Wallstrasse 40, 23560 Lübeck, Germany
email: jakoby / schindel @ informatik.mu-luebeck.de

The complexity classes Nearly-$\mathcal{BPP}$ and Med*DisP had been recently proposed as limits of efficient computation (Yamakami 96, Schindelhauer 96). For both classes, a polynomial time bounded algorithm with bounded probabilistic error has to compute correct outputs for at least $1 - 1/n^{\omega(1)}$ of inputs of length n. We generalize this notion to general error probabilities and arbitrary complexity classes. For proving the intractability of a problem it is necessary to show that it cannot be computed within a given error bound or every input length. For this, we introduce a new error complexity class, where the error is only infinitively often bounded by the error function.

We identify sensible bounds for the error function and derive new diagonalizing techniques. Using these techniques we present time hierarchies of a new quality: We are able to show that there are languages computable in time T that a machine with asymptotically slower running time cannot predict within a smaller error than 1/2.

Further, we investigate two classical non recursive problems: the halting problem and the Kolmogorov complexity function. We give strict lower bounds proving that any heuristic algorithm that claims to solve one of these problems makes unrecoverable errors with constant probability.

Chemnitzer Informatik-Berichte

Chemnitzer Informatik-Berichte

ISSN 0947-5125

Herausgeber: Fakultät für Informatik, TU Chemnitz
Straße der Nationen 62, D-09111 Chemnitz