University of Basel

Evaluation of Regression Search and State Subsumption in Classical Planning

Bachelor's Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Artificial Intelligence
http://ai.cs.unibas.ch/

Examiner: Prof. Dr. Malte Helmert
Supervisor: Dr. Martin Wehrle and Salomé Simon

Andreas Thüring
a.thuering@stud.unibas.ch
2009-724-162

31.07.2015

Abstract

The objective of classical planning is to find a sequence of actions which begins in a given initial state and ends in a state that satisfies a given goal condition. A popular approach to solve classical planning problems is based on heuristic forward search algorithms. In contrast, regression search algorithms apply actions "backwards" in order to find a plan from a goal state to the initial state. Currently, regression search algorithms are somewhat unpopular, as the generation of partial states in a basic regression search often leads to a significant growth of the explored search space. To tackle this problem, *state subsumption* is a pruning technique that additionally discards newly generated partial states for which a more general partial state has already been explored.

In this thesis, we discuss and evaluate techniques of regression and state subsumption. In order to evaluate their performance, we have implemented a regression search algorithm for the planning system Fast Downward, supporting both a simple subsumption technique as well as a refined subsumption technique using a trie data structure. The experiments have shown that a basic regression search algorithm generally increases the number of explored states compared to uniform-cost forward search. Regression with pruning based on state subsumption with a trie data structure significantly reduces the number of explored states compared to basic regression.

Table of Contents

1 Introduction

Classical planning is a sub-field of artificial intelligence that is concerned with finding a sequence of actions for an acting agent whose behavior is governed by rules specified by a problem definition. In order to achieve this, the problem is transformed into a rigid mathematical setting, for example using the Planning Domain Definition Language PDDL, which models a problem by specifying state variables and operators[11].

A planning system takes such a problem definition and tries to find a plan, that is a sequence of actions carried out by the agent, which starts in the defined initial state and ends in a desired goal state. A common approach to find a plan is to use a progression search algorithm, i.e. an algorithm which starts at the initial state and iteratively applies operators which are deemed applicable, creating new intermediate states until a plan is found. In contrast, a regression search algorithm applies operators in a backwards fashion in order to find a plan from a partial goal state to the specified initial state.

As the search space of planning problems can grow quickly, the investigation of different search algorithms and optimizations remains an important research topic. A popular approach to classical planning in the recent years is the utilization of progression search algorithms with heuristics[3, 8–10]. In contrast, as stated by Eyerich and Helmert [5], the effectiveness of these algorithms have rendered research in regression search algorithms somewhat unpopular momentarily.

Yet, for regression, there is the work of Alcázar et al. [1] who have used regression in conjunction with reachability-based heuristics. Furthermore, Alcázar et al. [2] have examined bidirectional planners, that is the combination of progression and regression search as well as the impact of state subsumption on such algorithms. Eyerich and Helmert [5] have utilized bidirectional planners as well, using perimeter search and pattern database heuristics. Haslum et al. [6] have examined h^m and pattern database heuristics for regression search algorithms.

Search algorithms may employ pruning strategies in order to reduce the search space explored during a planning task. Due to the creation of partial states in regression, a pruning technique based on state subsumption seems promising in regression search. However, the computational cost of finding suitable states for subsumption quickly outruns any performance gained by pruning. Instead of using typical closed list implementations of planning systems in the form of hash tables, employing an additional data structure for performing a subsumption check might prove to be worth the additional memory requirements if it sufficiently speeds up subsumption.

In this thesis, we discuss the techniques of regression and state subsumption and in order to evaluate their performance we have implemented a regression search algorithm for the planning system Fast Downward developed by Helmert [7], supporting pruning with a simple subsumption technique as well as a refined subsumption technique using a trie data structure. The implemented techniques were evaluated using a benchmark procedure. Results showed that while regression search in general performs worse than uniform cost search with which it was compared, it has great potential in some domains, where regression search algorithms which used pruning via subsumption expanded the fewest states in order to find a solution of all evaluated algorithms. These advantages are somewhat offset by the fact that a simple subsumption implementation is so computationally demanding that problem coverage is greatly reduced, while pruning using a trie data structure for subsumption has shown to reach performance comparable to a regression search algorithm without any subsumption.

2

Planning in SAS+

This chapter generally follows the definitions provided by Alcázar et al. [1] with some slight modifications for convenience.

Definition 1 (SAS$^+$ Planning Task)**.** An SAS$^+$ planning task is a 4-tuple $P = \langle V, s_0, s_E, O\rangle$, where

- V is a finite set of state variables, where each variable $v \in V$ has a finite domain D_v.
- A *partial state* s is a variable assignment for each variable $v \in V$ from $D_v \cup \{u\}$, where $u \notin D_v$ represents an undefined variable assignment.
- The *value* or *variable assignment* of a specific variable $v \in V$ in a given partial state s is defined as $s[v] \in D_v \cup \{u\}$.
- Let s be a partial state. We define the *set of defined variables* in s as $vars(s) := \{v \in V \mid s[v] \neq u\}$.
- A *state* s is a partial state for which $vars(s) = V$.
- O is a finite set of operators, where each operator $o \in O$ is a tuple $o = \langle cond(o), \mathit{eff}(o)\rangle$. The partial states $cond(o)$ and $\mathit{eff}(o)$ represent the preconditions and effects of o.
- Operators have an action cost defined as $cost(o) \in \mathbb{R}_0^+$.
- s_0 is the initial state.
- s_E is the partial state that defines the goals.

Definition 2 (Applicability of operators)**.** Let $P = \langle V, s_0, s_E, O\rangle$ be an SAS^+ planning task.

- An operator $o \in O$ is *applicable* in a state s if for all $v \in vars(cond(o)) : cond(o)[v] = s[v]$.
- The state $s' := app(o, s)$, called the *successor* of s, results from the application of o in s and is identical to s except for all $v \in vars(\mathit{eff}(o))$ which get a new variable assignment $s'[v] = \mathit{eff}(o)[v]$.

- A *path* between two states s and s' is a sequence of operators $\langle o_0, \ldots, o_n \rangle$ created by the successive application of these operators in s such that

$$s' = app(o_n, \ldots app(o_1, app(o_0, s)) \ldots).$$

  The state s' is said to be *reachable* by s.

- A *plan* is a path between the initial state s_0 and the state

$$s' = app(o_n, \ldots app(o_1, app(o_0, s_0)) \ldots),$$

  where state s' complies with the partial goal state s_E, i.e. $s'[v] = s_E[v]$ for all $v \in vars(s_E)$.

2.1 Progression Search Algorithms

The following section on progression search algorithms is mainly based on the textbook by Russell and Norvig [11].

Definition 3 (Progression Search algorithm)**.** Let $P = \langle V, s_0, s_E, O \rangle$ be an SAS^+ planning task.

- *Progression search* is the process of looking for a sequence of operators that reach a goal state s_E, beginning with the initial state s_0.
- The *search space* is the graph which is generated by the subsequent application of all applicable operators on all intermediate states beginning from the initial state s_0. Vertices in the graph correspond to states and edges represent the applied operators.
- A *search node* is a vertex in the search space and is defined as a 4-tuple $n = \langle s, p, o, c \rangle$ where s is the state corresponding to the node n, p is the parent node in the search space which created this node, $o \in O$ is the operator which generated the node and c is the cost associated with the path from the node representing the initial state to the node n.
- The cost of a path $\langle o_0, \ldots, o_k \rangle$ is defined as $\sum_{i=0}^{k} cost(o_i)$. It is sometimes associated with the specific last search node n_k of said path and defined as $g(n_k)$.

The goal of optimal planning is to find a plan which is optimal, i.e. a plan that has minimal cost among all plans, while satisficing planning tries to find any not necessarily optimal plan[11].

2.1.1 Procedure of a Progression Search Algorithm

The basic principles of a progression search algorithm are presented in this section, influenced by the textbook on artificial intelligence by Russell and Norvig [11].

- Given a search node n, the process of *expansion* determines all applicable operators $o \in O$ for the state s that belongs to this node and applies them, thus generating a new set of search nodes n', each containing a state $s' = app(o, s)$ created by applying o in s. Every generated search node n' is a child node of the parent node n in the search space. Before the node n is expanded, the goal test is executed on the node to determine whether a plan has already been found.

- Each of the nodes generated by the expansion of n is a leaf node, that is a node without children. The *open list* or *frontier* L_o is defined as the set of leaf nodes available for expansion. Often, a priority queue is used as data structure for implementing the open list.
- A search algorithm uses a *search strategy* for choosing a candidate for further expansion from the open list. For the purpose of this explanation, L_o is some data structure which allows the insertion of search nodes, as well as an operation denoted $pop()$ which retrieves the search node to be expanded, according to the search strategy used.
- The process of expansion can generate nodes describing states which have already been created by previous expansions. In order to avoid exploring these redundant paths, search algorithms employ a *closed list* L_c which contains every previously expanded node. If the closed list contains a search node corresponding to a state s' which is identical in all variable assignments to the state s belonging to a search node n that was newly generated during expansion, search node n can be discarded for further expansion without affecting completeness of the algorithm. The closed list is often implemented as a hash table, allowing fast retrieval of duplicates.

The procedure of a progression search algorithm is illustrated as pseudo code in Algorithm 1.

2.1.2 Search Strategies

An important characteristic of a search algorithm is its search strategy, i.e. the mechanism by which it governs the next candidate for expansion from the open list. Since we only employ blind search algorithms in this work, heuristic search strategies are only touched briefly. For a more thorough introduction about different search strategies and their properties see Russell and Norvig [11] from which these definitions were taken.

- *Blind search algorithms* only use the information provided by the problem definition in order to find a plan. An important blind search algorithm is uniform cost search, which always expands the node n with minimal path cost $g(n)$ first. Uniform cost search guarantees optimality of the found plan, if such a plan exists.
- *Informed search algorithms* employ a *heuristic function* in order to determine the most promising node in the open list for expansion.

Algorithm 1 Procedure of a progression search algorithm

procedure SEARCH(SAS$^+$ instance $P = \langle V, s_0, s_E, O\rangle$, initial node n_0)
    $L_c \leftarrow \emptyset$ ▷ Initialize closed list
    $L_o.insert(n_0)$
    loop
        if L_0 is empty **then**
            return false
        end if
        $n \leftarrow L_o.pop()$ ▷ choose new node from open list according to search strategy
        if $n.state[v] = s_E[v]$ for all $v \in vars(s_E)$ **then** ▷ Goal test
            return solution
        end if
        $L_c \leftarrow L_c \cup \{n\}$
        ▷ Expansion
        for $o \in O$ **do**
            if $applicable(n.state, o)$ **then**
                $n_{\text{new}} \leftarrow \langle app(o, n.state), n, o, g(n) + cost(o)\rangle$ ▷ Generate a new search node n_{new}
                if no node $n_c \in L_c$ exists with $n_c.state = n_{\text{new}}.state$ **then**
                    $L_o.insert(n_{\text{new}})$
                end if
            end if
        end for
    end loop
end procedure

3

Regression Search

Let $P = \langle V, s_0, s_E, O\rangle$ be an SAS^+ planning task. The objective of regression search is to employ a search algorithm which finds a path from the partial goal state s_E to the initial state s_0. Regression search generally uses the same principles as progression search strategies elaborated in section 2.1, bare some key differences noted in this chapter.

Definition 4 (Regressability of operators)**.** Let $P = \langle V, s_0, s_E, O\rangle$ be an SAS^+ planning task.

- Let $o \in O$ be an operator. The *set of variables that occur only in a condition but not in an effect* is defined as $\mathit{cond\_only}(o) := \{v \in V \mid \mathit{vars}(\mathit{cond}(o)) \setminus \mathit{vars}(\mathit{eff}(o))\}$.
- We define as regressability the "reverse applicability" of an operator $o \in O$. An operator o is *regressable* in partial state s if
  (i) there exists a $v \in \mathit{vars}(\mathit{eff}(o))$ for which $s[v] = \mathit{eff}(o)[v]$,
  (ii) there exists no $v \in \mathit{vars}(\mathit{eff}(o)) \cap \mathit{vars}(s)$ for which $s[v] \neq \mathit{eff}(o)[v]$,
  (iii) $s[v] = \mathit{cond}(o)[v]$ for all $v \in \mathit{vars}(\mathit{cond\_only}(o)) \cap \mathit{vars}(s)$
- The resulting state $s' = \mathit{regr}(o, s)$ from the *regression application* of o in s is called the predecessor of s and is identical to s except
  (i) $s'[v] = u$ for all $v \in \mathit{vars}(\mathit{eff}(o)) \setminus \mathit{vars}(\mathit{cond}(o))$.
  (ii) $s'[v] = \mathit{cond}(o)[v]$ for all $v \in \mathit{vars}(\mathit{cond}(o))$.
- A *regression path* between two partial states s and s' is a sequence of operators $\langle o_0, \ldots, o_n\rangle$ created by the successive application of these operators such that $s' = \mathit{regr}(o_n, \ldots \mathit{regr}(o_1, \mathit{regr}(o_0, s)) \ldots)$.
- A *regression plan* is a path $s' = \mathit{regr}(o_n, \ldots \mathit{regr}(o_1, \mathit{regr}(o_0, s_E)) \ldots)$, i.e. a path that transitions from the partial goal state s_E to a state s' which complies with the initial state s_0. A regression plan is always identical to a inverted plan in which the same operators are applied in a forward fashion.

3.1 Subsumption

Subsumption, as presented by Alcázar et al. [2], is a pruning strategy deployed during state expansion in a regression search algorithm.

Definition 5 (Subsumption of states)**.** Let $P = \langle V, s_0, s_E, O\rangle$ be an SAS^+ planning task and let s and s' be two partial states. Partial state s subsumes s' ($s \sqsubseteq s'$) if $s[v] = s'[v]$ or $s[v] = u$ for all $v \in V$.
If s' is subsumed by s, then the set of regressable operators of partial state s' is a subset of the set of regressable operators of s.

3.1.1 Using Subsumption in a Regression Search Algorithm

Consider a regression search algorithm that creates search node n during state expansion. Let the partial state s which corresponds to the search node n be subsumed by the partial state s' belonging to a search node n' contained in the closed list. If the optimality of the search algorithm shall be maintained, node n may only be pruned if additionally $g(n) \geq g(n')$. In satisficing planning, search node n may already be pruned if only the subsumption condition is fulfilled.

Algorithm 2 Procedure of a regression search algorithm with subsumption

procedure SEARCH(SAS$^+$ instance $P = \langle V, s_0, s_E, O\rangle$)
 $L_o.insert(n_E)$ ▷ Initialize open list
 $L_c \leftarrow \emptyset$ ▷ Initialize closed list
 loop
  if L_0 is empty **then**
   return false
  end if
  $n \leftarrow L_o.pop()$ ▷ choose $n \in L_o$ according to search strategy
  if $n.state[v] = s_0[v]$ for all $v \in vars(n.state)$ **then** ▷ initial state test
   return solution
  end if
  $L_c \leftarrow L_c \cup \{n\}$
  ▷ Expansion
  for $o \in O$ **do**
   if $regressable(n.state, o)$ **then**
    $n_{\text{new}} \leftarrow \langle regr(o, n.state), n, o, g(n) + cost(o)\rangle$
    for $n_{\text{c}} \in L_c$ **do**
     if $n_{\text{c}}.state \not\sqsubseteq n_{\text{new}}.state \vee g(n_{\text{new}}) < g(n_{\text{c}})$ **then**
      $L_o.insert(n_{\text{new}})$ ▷ Add newly created state to open list
     end if
    end for
   end if
  end for
 end loop
end procedure

4 Implementation

In order to assess the performance of regression search, we have implemented regression search functionality into the planning software Fast Downward, utilizing the already provided uniform cost search module. While implementation of operator applicability determination and state successor generation are straight forward, state subsumption causes special challenges.

4.1 Naive Implementation of Subsumption

Following the definitions in section 3.1, in order to determine whether the closed list contains a search node which corresponds to a partial state that subsumes a given newly created state, a basic subsumption check algorithm in the worst case has to access at least one variable of the partial state of every search node contained in the closed list, in the case that there is no subsuming partial state to be found. Considering that in many planning domains search spaces grow large quickly, the performance implications render using a basic state subsumption algorithm prohibitive in most settings.

4.2 Implementation of State Subsumption Utilizing a Trie Data Structure

In order to speed up the subsumption check, we implemented a trie data structure that performs said subsumption check into the planning system Fast Downward. A trie is an implementation of the closed list using a tree based on a specific layout which promises a more efficient retrieval of subsuming states from the closed list in comparison to a simple implementation. The trie implementation described here is adapted for state subsumption from Edelkamp and Schrödl [4] who present tries in the domain of subset dictionaries. To avoid confusion, in the following section trie nodes are always referenced by the letter n, while search nodes belonging to the search algorithm that employs the trie data structure are signified by the letter w.

Definition 6 (Subsumption trie)**.** Let $P = \langle V, s_0, s_E, O\rangle$ be an SAS^+ planning task. A *subsumption trie* $T = \langle N, E\rangle$ is a tree, where N is a finite set of nodes of T and $E \subseteq N \times \bigcup D_v \cup \{u\} \times N$ is a set of edges.

- Each *edge* $e \in E$ is a 3-tuple $\langle n, a, n' \rangle$, where $n \in N$ describes the parent node, $a \in \bigcup D_v \cup \{u\}$ is the edge label corresponding to a variable assignment of a partial state and $n' \in N$ is the child node.
- *Inner nodes* store no further information about entries.
- *Leaf nodes* are nodes at the lowest level of the trie T. They have no child nodes and in contrast to inner nodes they hold a reference to a search node w.
- A *path* $p = \langle e_1, \ldots, e_n \rangle$ from the root node of T to a leaf node corresponds to a partial state s that belongs to a search node w which has been previously inserted into the trie. Each edge $e_i = \langle n, a_i, n' \rangle$ of p on trie level l corresponds to the i-th variable assignment $s[v_i] = a_i$ of s. The leaf node in which the path ends associates the partial state s defined by the sequence of edge labels with a specific search node w.

Given the definitions above, *entries* of a subsumption trie are search nodes. Each inserted search node w is uniquely identified by a sequence of edges connected by trie nodes reaching from the root node to a leaf node which describes the variable assignments of partial state s corresponding to w, while the leaf node of said path associates s with its search node w. A *lookup* in the subsumption trie T, given a search node w which corresponds to the partial state s is defined as a procedure which retrieves a search node w' describing partial state s' where s' subsumes s and $g(w) \geq g(w')$, if such an entry has been inserted in T previously. If a lookup is able to retrieve such a search node w', search node w can be pruned.

4.2.1 Insertion

In order to use a subsumption trie in a regression search algorithm, all states that are added to the closed list during search must be inserted into the subsumption trie. Insertion is done recursively as follows. It is assumed that the i-th child c_i of a trie node n can be accessed in array-like fashion $n[c_i]$.

Algorithm 3 Insertion, recursion entry point

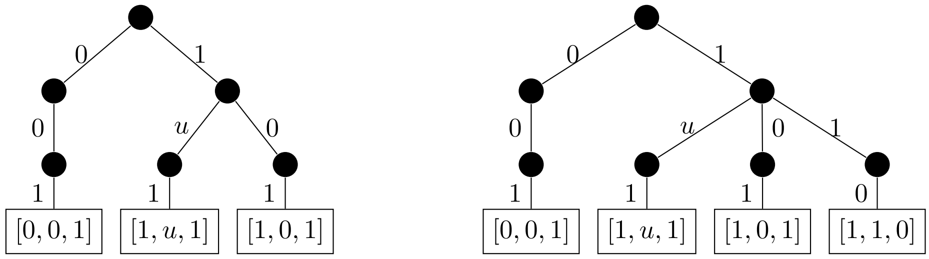
```
procedure INSERT(search node w, trie root r)
    INSERT(w, w.state, r, 0)
end procedure
```

The process of insertion is illustrated additionally in figure 4.1.

Algorithm 4 Insertion

```
procedure INSERT(search node w, partial state s, trie node n, index i)
    c ← s[i]
    if n[c] = ⊥ then                          ▷ Child node does not exist
        n[c] ← NODE                           ▷ Create new inner node
    end if
    if i = |s| − 1 then                       ▷ Search node n is a leaf node
        n.w ← w                               ▷ Insert reference to w in leaf node
        return
    else
        INSERT(w, s, n[c], i + 1)
    end if
end procedure
```

(a) Trie layout after insertion of state $[0, 0, 1]$ (b) Trie layout after insertion of state $[1, u, 1]$



(c) Trie layout after insertion of state $[1, 0, 1]$ (d) Trie layout after insertion of state $[1, 1, 0]$

Figure 4.1: Demonstration of Trie layout after insertion of four different states over three variables, each variable v having the same variable domain $D_v = \{0, 1\} \cup \{u\}$. Each path from the root node to a leaf represents a specific inserted partial state. In this figure, the stored partial state s is included in the leaf node for demonstration purposes. In an actual implementation of a subsumption trie a leaf node would contain a reference to the respective search node corresponding to the partial state described by the path in the trie that ends in said leaf node.

4.2.2 Lookup

The lookup procedure is likewise influenced by the algorithm provided by Edelkamp and Schrödl [4] and was slightly adopted. Given a search node w which corresponds to partial state s, the lookup algorithm starts by visiting the root of the trie and on every trie level l follows each edge which corresponds to the specific variable assignment $s[v_l]$. Additionally, any existing outgoing edges labeled u of a visited trie node will be followed as well. Thus, the lookup procedure will reach the leaf nodes of all previously inserted search nodes w' of which their respective partial states s' are identical to s, or which differ to s only in having one or more undefined variable assignments where s is defined, i.e. all partial states s' which subsumes s, as defined in section 3.1. Additionally, by retrieving the search node w' via the leaf node, path costs of w and w' are compared in order to determine whether the search node w can be pruned.

Algorithm 5 Lookup, recursion entry point

procedure LOOKUP(search node w, trie root r)
    return LOOKUP($w.state, r, 0$)
end procedure

Algorithm 6 Lookup

procedure LOOKUP(search node w, trie node n, index i)
    $s \leftarrow w.state$
    if $n["u"] \neq \perp$ **then** ▷ undefined child node exists
        if $i < |s| - 1$ **then** ▷ trie node n is an inner node
            if LOOKUP($w, n["u"], i + 1$) **then**
                return *true*
            end if
        else ▷ trie node n is a leaf node
            $w' \leftarrow n.w$ ▷ retrieve search node w' from leaf node
            if $g(w) \geq g(w')$ **then**
                return *true*
            end if
        end if
    end if
    $c \leftarrow s[i]$
    if $c \neq "u" \wedge n[c] \neq \perp$ **then** ▷ Specific child node $s[i]$ exists
        if $i < |s| - 1$ **then** ▷ trie node n is an inner node
            if LOOKUP($w, n[c], i + 1$) **then**
                return *true*
            end if
        else ▷ trie node n is a leaf node
            $w' \leftarrow n.w$ ▷ retrieve search node w' from leaf node n
            if $g(w) \geq g(w')$ **then**
                return *true*
            end if
        end if
    end if
    return *false*
end procedure

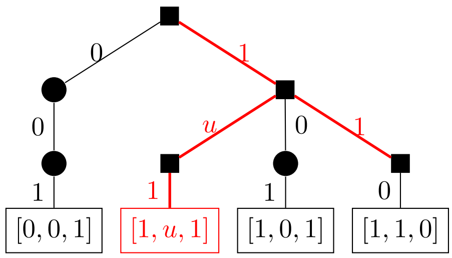


Figure 4.2: Lookup of state $[1, 1, 1]$ in the trie created by figure 4.1. The state $[1, u, 1]$ is returned as a possible candidate for subsumption.

4.2.3 Using a Subsumption Trie in the Context of a Search Algorithm

The following algorithm shows how a subsumption check for pruning can be integrated into a regression search algorithm as presented in Algorithm 2.

Algorithm 7 Procedure of a regression search algorithm with subsumption

```
procedure SEARCH(SAS⁺ instance P = ⟨V, s_0, s_E, O⟩, subsumption trie T)
    L_o.insert(n_E)                                          ▷ Initialize open list
    loop
        if L_0 is empty then
            return false
        end if
        n ← L_o.pop()
        if is_initial_state(n.state) then                    ▷ initial state test
            return solution
        end if
        T.insert(n)
                                                             ▷ Expansion
        for o ∈ O do
            if regressable(n.state, o) then
                n_new ← ⟨regr(o, n.state), n, o, g(n) + cost(o)⟩
                if T.lookup(n_new.state) = ⊥ then
                    L_o.insert(n_new)
                end if
            end if
        end for
    end loop
end procedure
```

5
Evaluation

In the following section we will present some findings about the performance implications of a progression search algorithm using subsumption, first using some theoretical considerations, and then evaluating the performance of the module we implemented in the planning system Fast Downward.

5.1 Subsumption Using a Priority Queue Implementation for the Closed List

Given a regression search algorithm with n search nodes contained in its closed list, each node representing a state with m variables having a domain of at most size l, and suggest an implementation of a subsumption technique using the usually provided closed list implementation of a planning system, which is in the form of a priority queue. Using such a data structure for subsumption, a search algorithm will in the worst case need to access nml memory locations and in any case at least one value of every node in the closed list needs to be examined if no partial state can be found which subsumes the given partial state. As n grows fast, the cost of pruning states based on simple subsumption quickly outruns the advantages gained from pruning.

5.2 Subsumption Using a Trie Implementation

A subsumption trie requires additional memory for the storage of its nodes, which might be prohibitive in some cases. The run time performance of a trie remains somewhat more elusive: it is strongly dependent on the branching factor of the trie, as during a lookup, each encountered edge with label u will cause the traversal of an additional sub-path. Edelkamp and Schrödl [4] estimate the complexity of a lookup in a trie as $O(n^{log(2-\frac{s}{m})})$, considering each entry has length m with $s \leq m$ variables undefined in a trie with n entries.

5.3 Experimentation

In this section we describe the results we obtained by implementing the proposed techniques for the planning system Fast Downward. Experiments were run on a cluster consisting of 2.66 *GHz* nodes and each experiment was run with a time limit of 30 minutes and a memory limit of 2048 *MB*. The performance of our implementation was compared to the uniform cost search algorithm already provided by the planning system, on which our implementations were based on. In this section, the uniform cost search module provided by Fast Downward is referenced by UCF, the basic regression algorithm without subsumption pruning is called regr, while the regression search algorithm with simple subsumption pruning is referenced by regr_s and the regression search algorithm with trie subsumption is named regr_T. As the number of evaluated problem domains was rather large, only a selection of results is presented here. In this chapter we will focus on the problem domains contained in the ipc11 competition as well as some domains where regression algorithms perform strongly. The full evaluation tables can be found in the appendix.

summary	**UCF**	**regr**	**regr$_s$**	**regr$_T$**
Coverage[1]	**521**	296	195	297
Expansions[2]	**1216.81**	14242.41	4418.32	4418.32
Memory[1]	3165260	4849984	**1354048**	1928504
Search time[2]	**0.02**	0.38	3.46	0.30

Table 5.1: Summary of evaluation results across all evaluated domains.

As shown in table 5.1, experiments revealed that regression search has been able to cover fewer problems than the UCF algorithm expect in the domains of *floortile*, *miconic* and *rovers*. Regression search without subsumption normally causes a great increase of the explored search space as compared to progression search: as seen in the same table, the geometric mean of expanded states in the evaluated domains is increased about tenfold in regr compared to UCF. This difference in expanded states is though highly problem dependent. The domains in which regr performs stronger than UCF are the domains where a regression algorithm generated less search nodes than a progression search algorithm. In some domains like *sokoban*, regression will create a great number of redundant search paths.

[1] Sum across all domains
[2] geometric mean across all domains

Coverage	UCF	regr	regr$_s$	regr$_T$
barman-opt11-strips (20)	**4**	0	0	0
elevators-opt11-strips (20)	**9**	2	0	2
floortile-opt11-strips (20)	2	**10**	2	9
logistics00 (29)	**11**	**11**	8	**11**
logistics98 (35)	**2**	**2**	**2**	**2**
miconic (150)	50	**60**	40	**60**
nomystery-opt11-strips (20)	**8**	7	4	7
openstacks-opt11-strips (20)	**15**	0	0	0
parcprinter-opt11-strips (20)	**6**	4	3	4
parking-opt11-strips (20)	**0**	**0**	**0**	**0**
pegsol-opt11-strips (20)	**17**	1	1	1
rovers (40)	5	**6**	4	5
satellite (36)	**5**	**5**	4	**5**
scanalyzer-opt11-strips (20)	**9**	5	3	5
sokoban-opt11-strips (20)	**18**	2	1	2
tidybot-opt11-strips (20)	**12**	0	0	0
transport-opt11-strips (20)	**6**	3	0	3
visitall-opt11-strips (20)	**9**	**9**	7	8
woodworking-opt11-strips (20)	**2**	0	0	0

Table 5.2: Coverage of the tested search algorithms in selected domains.

Using subsumption techniques, the number of expansions required by regression search algorithms in the evaluated domains is reduced to four times as many as are needed by UCF in the geometric mean which corresponds to about a third of the states expanded by regr, thus somewhat mitigating the complexity introduced by using regression. Yet, the computational demands of a simple subsumption method massively increase search time in all evaluated domains, rendering regr$_s$ the least efficient algorithm evaluated in this thesis.

Evaluation of the trie data structure shows that it can greatly mitigate the complexity introduced by using a subsumption check: Search times were reduced to a level comparable to regr, leading to a slightly better coverage. Still, even using a trie data structure for subsumption did not lead to a significant increase in performance compared to regression without any pruning based in subsumption. Yet, as seen in table 5.3, regression search algorithms with subsumption pruning will expand the fewest search nodes of all evaluated search algorithms in a handful of domains. In these domains regr$_T$ attained a coverage similar to and in some cases even slightly better than regr, suggesting that utilizing efficient data structures to perform pruning based on state subsumption can mitigate the inefficiency introduced by a simple subsumption pruning technique and prove to be a desirable approach.

Expansions	**UCF**	**regr**	**regr$_s$**	**regr$_T$**
floortile-opt11-strips (2)	13272508.87	100029.48	**59578.68**	**59578.68**
logistics00 (8)	**4621.56**	10942.18	8082.26	8082.26
logistics98 (2)	201674.32	787588.84	**101541.25**	**101541.25**
miconic (40)	2350.11	**1001.97**	**1001.97**	**1001.97**
nomystery-opt11-strips (4)	**6349.29**	33284.82	32791.80	32791.80
parcprinter-opt11-strips (3)	**2955.64**	44968.20	21954.74	21954.74
pegsol-opt11-strips (1)	**252.00**	19105.00	19105.00	19105.00
rovers (4)	1054.59	588.69	**341.24**	**341.24**
satellite (4)	5021.00	6933.10	**3433.16**	**3433.16**
scanalyzer-opt11-strips (3)	**4831.79**	5002.88	5002.88	5002.88
sokoban-opt11-strips (1)	**649.00**	5840751.00	1182.00	1182.00
transport-opt08-strips (5)	**466.25**	13898.89	7183.64	7183.64
visitall-opt11-strips (7)	**336.27**	832.33	750.22	750.22
woodworking-opt08-strips (3)	**2211.69**	117744.53	41206.91	41206.91

Table 5.3: Expansions needed to find a plan in selected domains. Each table entry gives the geometric mean of expansions for that domain

The values of used memory given in table 5.4 suggest that in most cases regression algorithm can profit from using subsumption memory-wise, as fewer states are expanded. As expected, using a trie for subsumption increases used memory, yet in the mean only by about a factor of 1.5 compared to regr$_s$(see table 5.1), as some space is saved by the efficient structure of a trie for some domains.

Memory	**UCF**	**regr**	**regr$_s$**	**regr$_T$**
floortile-opt11-strips (2)	1851552	21036	**18176**	39484
gripper (4)	**23388**	37428	26140	36680
logistics00 (8)	**49264**	60496	58860	73596
logistics98 (2)	36856	119580	**30720**	45904
miconic (40)	448684	**230892**	230896	291088
nomystery-opt11-strips (4)	**23216**	33328	33180	44804
parcprinter-08-strips (7)	**35992**	61752	46636	96640
parcprinter-opt11-strips (3)	**16136**	35176	24520	67032
pegsol-opt11-strips (1)	**5128**	21188	21188	29780
rovers (4)	19996	19720	**19588**	19980
satellite (4)	41404	54692	**30416**	39316
scanalyzer-opt11-strips (3)	**19304**	19356	19348	25168
sokoban-opt11-strips (1)	**5524**	589844	5660	9948
visitall-opt11-strips (7)	**41992**	49440	47008	66944
woodworking-opt08-strips (3)	**16164**	59152	35108	58680

Table 5.4: Memory usage of different search algorithms in selected domains. Each table entry gives the geometric mean of memory usage for that domain

In conclusion, while regression search algorithms in general perform worse than UCF, they remain a viable approach in specific domains. Pruning via subsumption is deeply dependent on the method used: a naive implementation introduces a complexity far too great to be useful, while a trie data structure achieves a performance comparable to regression search without subsumption. As the performance gain of pruning by state subsumption is highly dependent on the efficiency of the procedure that finds pruning candidates, further research in subsumption techniques may lead to more efficient methods that increase performance of regression search algorithms using pruning via subsumption compared to regression algorithms with no subsumption pruning, especially in domains that are already efficiently tackled by regression search algorithms.

6 Conclusion

We discussed the conceptual aspects of regression search algorithms in classical planning with special regard to pruning techniques based on s subsumption of partial states. To evaluate the performance of said techniques, a regression search algorithm was implemented into the planning system Fast Downward, supporting no subsumption pruning, subsumption pruning with a simple implementation as well as a refined technique using a trie data structure for subsumption detection.
The implementation was benchmarked in various problem domains. Results have shown that while regression search in general performs worse than uniform cost search, it is preferable in some domains which suit the regression paradigm well, expanding fewer nodes during the search process. Although pruning strategies based on subsumption generally greatly decrease the explored search space of a regression search, their performance is dependent on the underlying data structure used for performing the subsumption check. A simple subsumption strategy based on a hash table is so inefficient that problem coverage is reduced significantly compared to using no subsumption at all. Yet, evaluation of a subsumption technique relying on a trie data structure has shown to lead to performance comparable to a regression search algorithm with no subsumption pruning.

6.1 Future Work

Further research in efficient data structure for subsumption pruning may lead to additional performance gains, particularly as in our evaluation the search algorithms based on regression with subsumption pruning expanded the fewest search nodes overall. Integrating efficient subsumption pruning into heuristic regression search algorithms may also increase their performance.

Bibliography

[1] Vidal Alcázar, Daniel Borrajo, Susana Fernández, and Raquel Fuentetaja. Revisiting regression in planning. In *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, 2013. URL http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6877.

[2] Vidal Alcázar, Susana Fernández, and Daniel Borrajo. Analyzing the impact of partial states on duplicate detection and collision of frontiers. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling, ICAPS 2014, Portsmouth, New Hampshire, USA, June 21-26, 2014*, 2014. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS14/paper/view/7861.

[3] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artif. Intell.*, 129(1-2):5–33, 2001. doi: 10.1016/S0004-3702(01)00108-4. URL http://dx.doi.org/10.1016/S0004-3702(01)00108-4.

[4] Stefan Edelkamp and Stefan Schrödl. *Heuristic Search - Theory and Applications.* Academic Press, 2012. ISBN 978-0-12-372512-7. URL http://www.elsevierdirect.com/product.jsp?isbn=9780123725127.

[5] Patrick Eyerich and Malte Helmert. Stronger abstraction heuristics through perimeter search. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling, ICAPS 2013, Rome, Italy, June 10-14, 2013*, 2013. URL http://www.aaai.org/ocs/index.php/ICAPS/ICAPS13/paper/view/6031.

[6] Patrik Haslum, Blai Bonet, and Hector Geffner. New admissible heuristics for domain-independent planning. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, pages 1163–1168, 2005. URL http://www.aaai.org/Library/AAAI/2005/aaai05-184.php.

[7] Malte Helmert. The fast downward planning system. *CoRR*, abs/1109.6051, 2011. URL http://arxiv.org/abs/1109.6051.

[8] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What's the difference anyway? In *Graph Search Engineering, 29.11. - 04.12.2009*, 2009. URL http://drops.dagstuhl.de/opus/volltexte/2010/2432/.

[9] Malte Helmert and Hector Geffner. Unifying the causal graph and additive heuristics. In *Proceedings of the Eighteenth International Conference on Automated Planning and*

Scheduling, ICAPS 2008, Sydney, Australia, September 14-18, 2008, pages 140–147, 2008. URL http://www.aaai.org/Library/ICAPS/2008/icaps08-018.php.

[10] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *CoRR*, abs/1106.0675, 2011. URL http://arxiv.org/abs/1106.0675.

[11] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010. ISBN 978-0-13-207148-2. URL http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0136042597,00.html.

A
Appendix

A.1 Detailed Evaluation Results

coverage

Coverage	**UCF**	**regr**	**regr$_s$**	**regr$_T$**
airport (50)	**21**	16	11	11
barman-opt11-strips (20)	**4**	0	0	0
blocks (35)	**18**	9	6	9
depot (22)	**4**	1	1	1
driverlog (20)	**7**	6	3	6
elevators-opt08-strips (30)	**11**	4	0	4
elevators-opt11-strips (20)	**9**	2	0	2
floortile-opt11-strips (20)	2	**10**	2	9
freecell (80)	**15**	1	0	1
grid (5)	**1**	0	0	0
gripper (20)	**7**	6	4	**7**
logistics00 (29)	**11**	**11**	8	**11**
logistics98 (35)	**2**	**2**	**2**	**2**
miconic (150)	50	**60**	40	**60**
mprime (35)	**19**	5	1	8
mystery (30)	**15**	7	3	10
nomystery-opt11-strips (20)	**8**	7	4	7
openstacks-opt08-strips (30)	**20**	5	2	5
openstacks-opt11-strips (20)	**15**	0	0	0
openstacks-strips (30)	**7**	5	5	5
parcprinter-08-strips (30)	**10**	8	7	8
parcprinter-opt11-strips (20)	**6**	4	3	4
parking-opt11-strips (20)	**0**	**0**	**0**	**0**
pathways-noneg (30)	**4**	**4**	3	**4**
pegsol-08-strips (30)	**27**	8	3	6

pegsol-opt11-strips (20)	**17**	1	1	1
pipesworld-notankage (50)	**14**	1	1	1
pipesworld-tankage (50)	**11**	2	2	2
psr-small (50)	**49**	41	35	43
rovers (40)	5	**6**	4	5
satellite (36)	**5**	**5**	4	**5**
scanalyzer-08-strips (30)	**12**	8	6	8
scanalyzer-opt11-strips (20)	**9**	5	3	5
sokoban-opt08-strips (30)	**21**	5	4	5
sokoban-opt11-strips (20)	**18**	2	1	2
tidybot-opt11-strips (20)	**12**	0	0	0
tpp (30)	**6**	5	5	5
transport-opt08-strips (30)	**11**	8	5	8
transport-opt11-strips (20)	**6**	3	0	3
trucks-strips (30)	**6**	3	2	4
visitall-opt11-strips (20)	**9**	**9**	7	8
woodworking-opt08-strips (30)	**7**	4	3	5
woodworking-opt11-strips (20)	**2**	0	0	0
zenotravel (20)	**8**	7	4	7
Sum (1397)	**521**	296	195	297

The last row reports the sum across all domains.

expansions

Expansions	**UCF**	**regr**	**regr$_s$**	**regr$_T$**
airport (11)	**94.48**	201.66	189.16	189.16
blocks (6)	**187.76**	14894.56	6739.16	6739.16
depot (1)	**400.00**	13632.00	2306.00	2306.00
driverlog (3)	**5250.16**	42181.02	22811.79	22811.79
floortile-opt11-strips (2)	13272508.87	100029.48	**59578.68**	**59578.68**
gripper (4)	**4327.33**	19505.51	7007.63	7007.63
logistics00 (8)	**4621.56**	10942.18	8082.26	8082.26
logistics98 (2)	201674.32	787588.84	**101541.25**	**101541.25**
miconic (40)	2350.11	**1001.97**	**1001.97**	**1001.97**
mprime (1)	**197.00**	30489.00	2402.00	2402.00
mystery (5)	**15.71**	305.24	95.12	95.12
nomystery-opt11-strips (4)	**6349.29**	33284.82	32791.80	32791.80
openstacks-opt08-strips (2)	**400.90**	26194.85	26194.85	26194.85
openstacks-strips (5)	**4910.69**	21442.04	21442.04	21442.04
parcprinter-08-strips (7)	**413.33**	5370.47	2883.37	2883.37
parcprinter-opt11-strips (3)	**2955.64**	44968.20	21954.74	21954.74
pathways-noneg (3)	3681.45	2199.77	**1158.60**	**1158.60**
pegsol-08-strips (3)	**87.69**	1591.78	1591.78	1591.78
pegsol-opt11-strips (1)	**252.00**	19105.00	19105.00	19105.00
pipesworld-notankage (1)	**133.00**	1029125.00	30067.00	30067.00
pipesworld-tankage (2)	**367.11**	80160.16	8776.16	8776.16
psr-small (35)	**186.34**	13098.82	2566.72	2566.72
rovers (4)	1054.59	588.69	**341.24**	**341.24**
satellite (4)	5021.00	6933.10	**3433.16**	**3433.16**
scanalyzer-08-strips (6)	**1561.37**	1644.41	1644.41	1644.41
scanalyzer-opt11-strips (3)	**4831.79**	5002.88	5002.88	5002.88
sokoban-opt08-strips (4)	**1157.62**	1896665.09	4983.11	4983.11
sokoban-opt11-strips (1)	**649.00**	5840751.00	1182.00	1182.00
tpp (5)	**194.73**	323.75	287.02	287.02
transport-opt08-strips (5)	**466.25**	13898.89	7183.64	7183.64
trucks-strips (2)	**11763.80**	97303.25	16128.81	16128.81
visitall-opt11-strips (7)	**336.27**	832.33	750.22	750.22
woodworking-opt08-strips (3)	**2211.69**	117744.53	41206.91	41206.91
zenotravel (4)	**202.05**	1181.41	1016.72	1016.72
Geometric mean (197)	**1216.81**	14242.41	4418.32	4418.32

Only instances where all configurations have a value for ”expansions” are considered. Each table entry gives the geometric mean of ”expansions” for that domain. The last row reports the geometric mean across all domains.

memory

Memory	**UCF**	**regr**	**regr$_s$**	**regr$_T$**
airport (11)	**57468**	57864	57600	78176
blocks (6)	**29584**	42896	38740	45984
depot (1)	**4996**	6004	5260	5516
driverlog (3)	**19040**	28672	24756	30972
floortile-opt11-strips (2)	1851552	21036	**18176**	39484
gripper (4)	**23388**	37428	26140	36680
logistics00 (8)	**49264**	60496	58860	73596
logistics98 (2)	36856	119580	**30720**	45904
miconic (40)	448684	**230892**	230896	291088
mprime (1)	**5128**	19252	6840	7252
mystery (3)	**16048**	54816	20816	23176
nomystery-opt11-strips (4)	**23216**	33328	33180	44804
openstacks-opt08-strips (2)	**9860**	13528	13528	20888
openstacks-strips (5)	**25744**	29816	29816	38512
parcprinter-08-strips (7)	**35992**	61752	46636	96640
parcprinter-opt11-strips (3)	**16136**	35176	24520	67032
pathways-noneg (3)	18920	19980	**18516**	36036
pegsol-08-strips (3)	**15120**	23444	23444	31632
pegsol-opt11-strips (1)	**5128**	21188	21188	29780
pipesworld-notankage (1)	**5128**	560552	36944	51612
pipesworld-tankage (2)	**10256**	33364	14384	29020
psr-small (35)	**172240**	1412524	236300	335236
rovers (4)	19996	19720	**19588**	19980
satellite (4)	41404	54692	**30416**	39316
scanalyzer-08-strips (6)	**36772**	36832	36824	45600
scanalyzer-opt11-strips (3)	**19304**	19356	19348	25168
sokoban-opt08-strips (4)	**20784**	958584	23096	36144
sokoban-opt11-strips (1)	**5524**	589844	5660	9948
tpp (5)	**25672**	38688	34244	59224
transport-opt08-strips (5)	**26012**	45784	40932	50880
trucks-strips (2)	**11360**	22532	13576	19892
visitall-opt11-strips (7)	**41992**	49440	47008	66944
woodworking-opt08-strips (3)	**16164**	59152	35108	58680
zenotravel (4)	**20528**	31772	30988	37708
Sum (195)	3165260	4849984	**1354048**	1928504

Only instances where all configurations have a value for "memory" are considered. Each table entry gives the sum of "memory" for that domain. The last row reports the sum across all domains.

search_time

Search time	**UCF**	**regr**	**regr$_s$**	**regr$_T$**
airport (11)	**0.01**	0.02	0.09	0.13
blocks (6)	**0.01**	0.12	2.39	0.15
depot (1)	**0.01**	0.16	0.40	0.10
driverlog (3)	**0.04**	0.44	25.13	0.34
floortile-opt11-strips (2)	65.13	**1.55**	147.89	2.00
gripper (4)	**0.02**	0.16	1.64	0.10
logistics00 (8)	**0.03**	0.15	8.83	0.18
logistics98 (2)	**0.85**	11.31	886.55	2.89
miconic (40)	**0.04**	0.05	0.22	0.05
mprime (1)	**0.01**	1.98	5.40	0.27
mystery (5)	**0.01**	0.12	0.29	0.05
nomystery-opt11-strips (4)	**0.02**	1.18	21.72	1.23
openstacks-opt08-strips (2)	**0.01**	0.21	17.33	0.49
openstacks-strips (5)	**0.01**	0.26	10.90	0.44
parcprinter-08-strips (7)	**0.01**	0.09	1.09	0.20
parcprinter-opt11-strips (3)	**0.03**	0.40	28.21	1.97
pathways-noneg (3)	**0.03**	0.07	0.34	0.09
pegsol-08-strips (3)	**0.01**	0.06	1.09	0.08
pegsol-opt11-strips (1)	**0.01**	0.74	743.96	1.38
pipesworld-notankage (1)	**0.01**	71.02	1347.22	24.47
pipesworld-tankage (2)	**0.01**	1.17	10.23	0.91
psr-small (35)	**0.01**	0.13	0.49	0.08
rovers (4)	**0.01**	**0.01**	**0.01**	**0.01**
satellite (4)	**0.06**	0.14	0.83	0.09
scanalyzer-08-strips (6)	**0.04**	0.15	0.61	0.15
scanalyzer-opt11-strips (3)	**0.07**	0.31	2.42	0.31
sokoban-opt08-strips (4)	**0.01**	25.01	4.73	2.06
sokoban-opt11-strips (1)	**0.01**	222.91	1.29	4.99
tpp (5)	**0.01**	0.03	0.09	0.03
transport-opt08-strips (5)	**0.01**	0.23	2.68	0.17
trucks-strips (2)	**0.02**	2.41	10.68	0.56
visitall-opt11-strips (7)	**0.02**	0.04	0.26	0.04
woodworking-opt08-strips (3)	**0.02**	3.89	97.65	2.45
zenotravel (4)	**0.01**	0.14	1.21	0.13
Geometric mean (197)	**0.02**	0.38	3.46	0.30

Only instances where all configurations have a value for "search_time" are considered. Each table entry gives the geometric mean of "search_time" for that domain. The last row reports the geometric mean across all domains.

Declaration on Scientific Integrity
Erklärung zur wissenschaftlichen Redlichkeit

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

Author — Autor
Andreas Thüring

Matriculation number — Matrikelnummer
2009-724-162

Title of work — Titel der Arbeit
Evaluation of Regression Search and State Subsumption in Classical Planning

Type of work — Typ der Arbeit
Bachelor's Thesis

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, 31.07.2015