Artificial Intelligence 173 (2009) 1310–1342

Contents lists available at ScienceDirect

Artificial Intelligence

www.elsevier.com/locate/artint

Weighted A* search – unifying view and application

Rüdiger Ebendt [a,*], Rolf Drechsler [b]

[a] German Aerospace Center, Institute of Transportation Systems, 12489 Berlin, Germany
[b] Institute of Computer Science, University of Bremen, 28359 Bremen, Germany

ARTICLE INFO

Article history:
Received 17 December 2007
Received in revised form 9 June 2009
Accepted 10 June 2009
Available online 16 June 2009

Keywords:
Planning
Search
Heuristic search
A*
Weighted A*
BDD
STRIPS

ABSTRACT

The A* algorithm is a well-known heuristic best-first search method. Several performance-accelerated extensions of the exact A* approach are known. Interesting examples are approximate algorithms where the heuristic function used is inflated by a weight (often referred to as weighted A*). These methods guarantee a bounded suboptimality.

As a technical contribution, this paper presents the previous results related to weighted A* from authors like Pohl, Pearl, Kim, Likhachev and others in a more condensed and unifying form. With this unified view, a novel general bound on suboptimality of the result is derived. In the case of avoiding any reopening of expanded states, for $\epsilon > 0$, this bound is $(1+\epsilon)^{\lfloor \frac{N}{2} \rfloor}$ where N is an upper bound on an optimal solution length.

Binary Decision Diagrams (BDDs) are well-known to AI, e.g. from set-based exploration of sparse-memory and symbolic manipulation of state spaces. The problem of exact or approximate BDD minimization is introduced as a possible new challenge for heuristic search. Like many classical AI domains, this problem is motivated by real-world applications. Several variants of weighted A* search are applied to problems of BDD minimization and the more classical domains like blocksworld and sliding-tile puzzles. For BDD minimization, the comparison of the evaluated methods also includes previous heuristic and simulation-based methods such as Rudell's hill-climbing based sifting algorithm, Simulated Annealing and Evolutionary Algorithms.

A discussion of the results obtained in the different problem domains gives our experiences with weighted A*, which is of value for the AI practitioner.

1. Introduction

In many real-world problems, dominating effort is put into search which often involves huge state spaces. Therefore a large number of papers on search has been published by numerous authors. The drawbacks of fixed-order methods like breadth-first search or depth-first search can be avoided by following a best-first order. The disadvantages of blind methods are overcome by heuristic search methods that guide the search. A prominent guided best-first search algorithm is the well-known A* algorithm [40]. Best-first search is a more general framework of algorithms, e.g. see [55]: besides A*, other examples for special cases of best-first search are breadth-first search and Dijkstra's single-source shortest path algorithm [22]. Best-first search explores the search graph by a list OPEN containing the "open" frontier nodes that have been generated but not yet expanded. A second list CLOSED stores the "closed" inner or expanded nodes. A cost function maps every node to its cost value. A best-first search always expands a most promising open node of minimum cost. Expanding a node means to generate all its child nodes. They are inserted into OPEN, preserving an order based upon the cost values of the nodes.

* Corresponding author.
E-mail addresses: ruediger.ebendt@dlr.de (R. Ebendt), drechsle@informatik.uni-bremen.de (R. Drechsler).

0004-3702/$ – see front matter
doi:10.1016/j.artint.2009.06.004

The expanded node is inserted into CLOSED. At the start, OPEN contains only the initial node and the search stops when a goal node is chosen for expansion.

The different instances of best-first search differ only in their cost functions. For A*, the cost of a node n is $f(n) = g(n) + h(n)$. Hereby, two components of information are used with every node n: one is $g(n)$, which is the information about the cost of the path already covered. The other is the heuristic function value $h(n)$, which is an estimate of the least cost of the remaining part of the path to a goal node. The vertices of the search graph represent the states of a problem state space, e.g. for the sliding-tile puzzle, a state is represented by an ordered sequence of tiles. The edges of the search graph describe the possible transitions between the states (for more details, see Section 2.1).

A* is used in many fields of application, including diverse areas such as robotics (see e.g. [62–64]), computational biology (see e.g. [51,59,78,88]), AI gameplay (see e.g. [12]), hardware verification [75] and logic synthesis [28]. If certain requirements to the heuristic function guiding the search are met, A* will find a minimum cost path to a goal state (see [40] and also Section 2.1).

A serious drawback of A* is that, in the worst-case, the run time as well as the amount of memory required is exponential in the depth of the search. This has led to several extensions of A*, some of which are memory-bounded [13,49, 59,77,87,89,90], while others mainly aim at a reduction in run time by allowing for bounded suboptimality. These so-called weighted A* methods actually do guarantee bounded suboptimality which contrasts to generalizations of best-first search such as the K-Best First Method (KBFM) of Felner et al. [33]. Generally, a weighting to the heuristic function is applied and the methods differ in the way and idea of this weighting.

First, Pohl proposed to constantly inflate the heuristic function h by a certain factor $1 + \epsilon$, $\epsilon > 0$ [71]. Since this is the original idea of weighted A*, many authors refer to it as WA*. A second proposal of the same author was *Dynamic Weighting* (DWA*) [72]. At the beginning of the search, the method starts with a high weighting of the heuristic function (as this may help to find a promising direction more quickly) and then it dynamically weights it less heavily as the search goes deeper. The latter may help to prevent premature termination. More recently, the interest in Pohl's first, conceptually simpler and more natural idea of a constant overweighting has been renewed [52,88]. It has also been embedded in the *Anytime Weighting* A* (AWA*) [38] and *Anytime Repairing* A* (ARA*) variants of A* in [62,63].

In contrast to that, in [70], the *Traveling Salesman Problem* (TSP) has been tackled by an extension of A* called A^*_ϵ which relaxes the *selection condition* of A*. This condition triggers the choice of the next node for expansion (i.e. for generating all its child nodes). Further, the idea of relaxing a (unidirectional) A* search has also been transferred to the bidirectional case [53].[1]

In this paper, previous results from the different authors are presented in a condensed and unifying form. This allows for a comparison of the respective methods on more formal grounds. Finally, a novel general bound on suboptimality is derived (see Section 5). It is shown that all discussed variants of weighted A* remain methods of *bounded* suboptimality, even when expanded states are not reopened again. A previous bound was only stated for the case of constant overweighting [63].

A second contribution of this paper is the experimental evaluation of the several variants. In Section 7, the respective variants of weighted A* (two of them are novel) are applied to benchmark problems of classical AI domains like blocks-world, several logistics domains, the sliding-tile puzzle, and a problem domain which has been well-studied in the hardware community. This is the problem of the exact or approximate minimization of *Binary Decision Diagrams* (BDDs). BDDs are well-known in the AI as well as in the hardware community, e.g. from set-based exploration of sparse-memory and symbolic manipulation of state spaces. For this domain, the comparison of the evaluated methods also includes previous heuristic and simulation-based methods such as Rudell's hill-climbing based sifting algorithm [76], *Evolutionary Algorithms* (EAs) [23], and *Simulated Annealing* (SA) [4].

BDDs uniquely represent Boolean functions and are described in more detail in Section 6.3. It has been shown that it is NP-complete to decide whether the number of nodes of a BDD can be decreased by variable reordering [5]. Although the solving of NP-complete problems has a long tradition in the AI community (e.g. the *Traveling Salesman Problem* (TSP) [70], the $(n \times n) - 1$ sliding-tile puzzle [60,74], number partitioning [56], rectangle and bin packing [57,58], minimum vertex cover [37,61] and many of the planning problems [11,54]), and BDDs have been used in the community for sparse-memory exploration [44,89] and to run symbolic versions of established search methods like A* [30,39,43,73], to the best of the authors' knowledge, so far the problem of exact or approximate BDD minimization itself has not been addressed in AI. However, this problem is strongly motivated by real-world applications in VLSI CAD. Therefore the authors would also like to introduce this problem to AI as a possible new challenge. A discussion of the experimental results obtained in the different problem domains addresses the AI practitioner.

This paper is structured as follows: A brief description of A*, basic notations and definitions are given and the different ideas of weighted A* are briefly reviewed. Related technical results of different authors are presented in a unifying form in Section 3. In Section 4, an instructive example illustrates the reopening of expanded nodes for weighted A*. The consequences of *not* reopening the nodes are discussed in Section 5. In Section 6, the problem of exact and approximate BDD minimization is introduced as a possible new challenge for weighted A*. For this purpose, first a formal definition of BDDs and the problem is given. Then previous work on exact BDD minimization by A* is briefly reviewed. Experimental results are given in Section 7. Finally, in Section 8 the work is concluded.

[1] In addition, [48] gives an excellent discussion of the relationship of unidirectional to bidirectional search in general.

2. Background

2.1. State space search by A* algorithm

A search problem and the corresponding search task can be formulated as a state space graph. Vertices represent the states q and edges represent the allowed state transitions (see e.g. [55] and also Section 1). For the remainder of the paper, vertices will be identified with the states they represent and the letters q, q', ..., which are traditionally used to denote states, are favored over the letters n, n', ... that are usually used for the (logical) nodes of the search graph.

An important method to *guide* the search on a state space is *heuristic search*. With every state q a quantity $h(q)$ is associated which estimates the cost of the cheapest path from q to a goal state t (the "*target*"). This allows for searching in the direction of the goal states. The A* algorithm is a heuristic best-first search algorithm (see also Section 1). Given that certain requirements are met, a minimum cost path from the initial state s (the "*start*") to a goal state t is found by A*. It starts at s and bases the choice of the next state to expand on two criteria:

- the sum of the edge costs on the path from the initial state up to state q (or "path cost" for short), denoted $g(q)$, and
- the estimate $h(q)$.

They are combined in cost function $f = g + h$. For all q the cost of q is the sum of its current path cost and its estimate, i.e. we have $f(q) = g(q) + h(q)$. The cost of a minimum cost path from s to q is denoted $g^*(q)$. The cost of a minimum cost path from q to a goal state is denoted $h^*(q)$.

For A*, the estimate $h(q)$ has to be a lower bound on the cost of an optimal path from q to a goal state:

$$h(q) \leqslant h^*(q) \tag{1}$$

In this case, h is called *admissible*. A* is called an *admissible algorithm* since the theory guarantees that A* terminates and always finds a minimum cost path [40]. A* is also known to be optimally efficient in terms of the number of expanded nodes (up to tie breaking) in a class of search algorithms that use the same heuristic [20].

Like every best-first search, A* maintains a prioritized queue Open which is ordered with respect to ascending values $f(q)$ (see also Section 1). Initially, this queue contains only s. At each step, a state q with a *minimal* f-value is expanded, dequeued and put on a list called Closed. During expansion, the successor states of q are generated and inserted into the queue Open according to their f-values. For this, the g- and the h-value of the successor states are computed dynamically. For a transition $q \longrightarrow q'$ let $c(q, q')$ denote the transition cost (edge cost). Then q' is associated with its cost $g(q') = g(q) + c(q, q')$, i.e. g accumulates transition costs. In this, for a state q, $g(q)$ is computed as the sum of the cost $c(r, r')$ of all transitions $r \longrightarrow r'$ occurring on the current tentative path to q. If a path between q and q' is optimal, its cost is denoted by $k(q, q')$.

If there is more than one state with a minimum f-value, *tie-breaking rules* are used to select one of these states. The most common rule is to select a state with a lowest h-value. Such states are less estimative and when a search from such a state is continued, a faster termination can be expected. This idea is used again later in all methods described in Sections 2.3.1, 2.3.2 and 2.3.3.

A successor state q' might be generated a second time if q' has more than one predecessor state. If a cheaper path from s to q' is found in this case, $g(q')$ is updated. If q' was on the list Closed, q' is *reopened*, i.e. it is put on Open again. Thus states get a second chance during the search for the minimum cost path when new information about them is available. These updates of the g-component of f to the costs of a newly found cheaper path continuously compensate for the fact that the character of the h-component is only estimative. The cheapest known path to q' is denoted $p(q')$ and is also updated, respectively.[2]

The algorithm terminates if the next state to expand is a goal state t. The estimate $h(t) = h^*(t)$ must be zero. In this case, the path found up to t is of minimal cost, which is denoted C^*, C^* as well as the optimal path (denoted $p^*(t)$) is reported as solution. As has been stated in Section 1, it is the cost function which separates A* from other best-first searches like Dijkstra's single-source shortest path algorithm ($f = g$ with g defined as above), greedy best-first search ($f = h$ with h defined as above) or breadth-first search, if the edge costs are uniform and, again, $f = g$.

2.2. Monotonicity

Definition 1. Consider an A* algorithm with heuristic function h (satisfying $h(t) = 0$ for all goal states t) and with the cost k of optimal paths between states. Heuristic function h is said to be *monotone* (or, equivalently, *consistent*), if

$$h(q) \leqslant k(q, q') + h(q') \tag{2}$$

for any descendant q' of q.

[2] In an implementation, usually only a back-pointer to the predecessor is stored. The complete path p can be reconstructed as the sequence of predecessors up to the initial state.

In [40] it is shown that, in the case of a monotone heuristic function h, A* finds optimal paths to all expanded nodes. More precisely:

Theorem 1. *Consider an* A* *algorithm with a monotone heuristic function h and path cost function g. Then, if a state q is expanded, a cheapest path to q has already been found, i.e. we have* $g(q) = g^*(q)$ *and therefore* $p(q) = p^*(q)$.

Cost updates are never needed after a first expansion. With that, every state is expanded exactly once and there is no performance degradation by the reopening of states (as described later in Section 4).

2.3. Weighted A* search

In the literature, the term "weighted A*" subsumes several approximate variants of A*. All approaches relax some of the conditions of A*. This is done to derive a faster algorithm with a provable upper bound on suboptimality. The ideas vary significantly and in the following three methods are distinguished.

2.3.1. Constant inflation

In [71], Pohl's first suggestion was the *constant* inflation of the heuristic function h by a fixed factor $1+\epsilon$ ($\epsilon > 0$). That is, the cost function

$$f^{\uparrow}(q) = g(q) + (1+\epsilon) \cdot h(q)$$

is used instead of the original cost function f of A*.

This method is denoted WA* since historically it is the first weighted variant of A* that has been proposed. When speaking of "weighted A*" in the sequel, we refer to the collection of all three methods, whereas WA* denotes the particular method suggested by Pohl.

Even if h is admissible, that would not always hold for the inflated heuristic (see e.g. [55]). The admissibility condition of A* is relaxed to quickly direct the search into a more promising direction. It is noteworthy that WA* breaks ties in favor of the state with the lower h-value (without establishing a special tie breaking rule). Given, that h is admissible, it can be shown that WA* is ϵ-*admissible*, i.e. it always finds a solution whose cost does not exceed the optimal cost by more than a factor of $1+\epsilon$. In comparison to the "dynamic" variant DWA* (see the next section), no other precautions against premature termination are taken here.

2.3.2. Dynamic weighting

Pohl's second idea [72] was to relax the fixed weighting of the heuristic function h. Algorithm DWA* starts with a high weighting of h at the beginning of the search. This may help to find a promising direction more quickly. Then the method dynamically weights the heuristic less heavily as the search goes deeper, preventing premature termination. For $\epsilon > 0$, the cost function used by DWA* is

$$f^{\mathrm{DW}}(q) = g(q) + h(q) + \epsilon \cdot \left[1 - \frac{d(q)}{N}\right] \cdot h(q)$$

where $d(q)$ denotes the depth of the vertex representing state q in the search graph, and N denotes the optimal solution length, respectively. Usually, N is not known in advance, but then an upper bound or an estimate can be used instead. For some problems however, N is actually known. For example, sometimes all paths to a goal vertex in this graph are of equal length, and thus the length N is the number of edges on such a path. The knowledge of N can be used for many improvements. It can be shown that the same ϵ-admissibility as for WA* (see previous section) holds for DWA*, if h is admissible. Both methods are further analyzed in Section 3.

2.3.3. Search effort estimates

This section reviews an important approach of Pearl and Kim [70]. The first significant point is the extension of a set of states crucial for A* to a larger set, thereby "blurring" the focus and relaxing the selection condition of A*. This set is the set of states with minimal f-value on OPEN. It is extended to the larger set of states with an f-value within $1+\epsilon$ times the least f-value. The second important proposal is the use of a heuristic to estimate the remaining search effort at a given snapshot of the search progress. This additional heuristic then selects a state from the aforementioned extended set of focus as the next state to expand. Thereby it provides an additional guide of the search process.

In detail, Pearl and Kim [70] suggest an extension of A* called A^*_ϵ which adds a second queue FOCAL maintaining a subset of the states on OPEN. This subset is the set of those states whose cost does not deviate from the minimal cost of a state on OPEN by a factor greater than $1+\epsilon$. More precisely,

$$\text{FOCAL} = \left\{ q \mid f(q) \leqslant (1+\epsilon) \cdot \min_{r \in \text{OPEN}} f(r) \right\} \tag{3}$$

The operation of A^*_ϵ is identical to that of A* except that A^*_ϵ selects a state q from FOCAL with minimal value $h_F(q)$. The function h_F is a second heuristic estimating the computational effort required to complete the search. In this, the nature of

h_F differs significantly from that of h since h estimates the *solution cost* of the remaining path whereas h_F estimates the remaining *time* needed to find this solution. The choice of h_F puts a high degree of freedom to the approach which will be subject to further investigation in Section 3. In [70], it has been suggested to use

- $h_F = h$ or
- to integrate properties of the subgraph emanating from a given state q.

The motivation behind the first point is that minimizing the h-value for the states in the set FOCAL means preferring the less estimative states. As a concrete suggestion for the second point,

$$h_F(q) = N - d(q)$$

will be used later in the experimental evaluation (see Section 7). Again, $d(q)$ denotes the depth of the vertex representing state q in the search graph and N is an upper bound on an optimal solution length. To minimize $N - d(q)$ means to prefer the deeper states in the search graph. This is done with the motivation that the subgraphs emanating from them tend to be comparatively small and thus the same can be expected for the remaining run time. If h is admissible, A^*_ϵ is ϵ-admissible, i.e. we have the same upper bound $1 + \epsilon$ on suboptimality as for WA* and DWA*.

3. Unifying view

In this section, the approach of Pearl and Kim [70] which has been reviewed as the method A^*_ϵ in the last section, is subjected to a closer consideration. Special attention is drawn to its nondeterministic formulation. Depending on the choice of the additional heuristic h_F, it basically allows for expansion of any state within the extended focus, i.e. within the set FOCAL. As will be shown, this allows for viewing of earlier methods as special cases of A^*_ϵ. Historically, A^*_ϵ has been published after the idea of constant inflation (WA*) and Dynamic Weighting (DWA*). This suggests that Pearl and Kim already developed their algorithm aiming at a generalization of the earlier approaches. In so far, the following unifying view has already been prepared by the work of Pearl and Kim. Nevertheless, in the following, the relationship of A^*_ϵ to WA* and DWA* is made explicit for the first time. This enables the transfer of the more general results to the special cases, and in turn allows for a comparison of the respective methods on more formal grounds. As the first step, the next result states a condition that guarantees the conformity of a cost function with the strategy described in Section 2.3.3.

Theorem 2. *Let us consider a state space with a subset* OPEN *and a cost function* f*, let* $\epsilon > 0$ *and let* FOCAL *be defined as before in Eq.* (3)*. For all states* q *of the state space, let* $f^{\Uparrow}(q) = (1 + \epsilon) \cdot f(q)$ *and let* $f(q) \leqslant f'(q) \leqslant f^{\Uparrow}(q)$*. Let*

$$\hat{q} = \arg\min_{q \in \text{OPEN}} f'(q)$$

Then it must be that $\hat{q} \in$ FOCAL*.*

Proof. See Appendix A. □

In Section 2.3.3 we stated that the choice of the heuristic function h_F leaves a considerable degree of freedom to the method. Next, we will clarify that the other relaxation methods can be derived simply by respective choices for h_F. In detail, Theorem 2 characterizes DWA* and WA* as two instantiations of the generic method given in Section 2.3.3. In this, Pearl and Kim's proposal proves to be more than just another weighted variation of A*. The next result shows that it also serves as a framework for weighted A*.

Theorem 3. *Let us consider a state space and its graph representation, let* g *be the path cost function and* h *be the heuristic function of* A*, and let* $\epsilon > 0$ *be the parameter of the* A*-variations* A^*_ϵ*,* WA*, and* DWA*, respectively. For all states* q *of the state space, let* $f^{\uparrow}(q) = g(q) + (1 + \epsilon) \cdot h(q)$*, let* $d(q)$ *denote the depth of the vertex representing* q*, let* N *denote an upper bound on an optimal solution length, and let* $f^{\text{DW}} = g(q) + h(q) + \epsilon \cdot [1 - \frac{d(q)}{N}] \cdot h(q)$*. Further, assume that identical tie-breaking rules are used in the algorithms. Then we have*:

- *the operation of Algorithm* WA* *is identical to that of* A^*_ϵ *with search estimate* $h_F = f^{\uparrow}$*, and*
- *the operation of Algorithm* DWA* *is identical to that of* A^*_ϵ *with search estimate* $h_F = f^{\text{DW}}$*.*

Proof. See Appendix A. □

In brief, the result states that the choice of the next state to expand as performed by DWA* and WA* conforms to the relaxation strategy of A^*_ϵ as stated in Eq. (3). Notice that, despite the fact that DWA* and WA* are formulated by use of cost functions that are *different* from that of A* or A^*_ϵ (i.e., different from $f = g + h$), they provably act as if $f = g + h$ would

be used. This holds since, in the formulation as an instantiation of A^*_ϵ, they are also guided by the second heuristic h_F. It is precisely this function h_F that then must be replaced by the respective alternative cost function.

The result of Theorem 3 also allows any provable result for A^*_ϵ to be transferred directly to DWA* and WA*, as the two methods are special instances of A^*_ϵ. For instance, it is well known that A^*_ϵ is ϵ-admissible. The same property follows for DWA* and WA* immediately. There is no need for specific proofs with respect to a particular instantiation of A^*_ϵ. It is also known that $f(q) \leqslant (1+\epsilon) \cdot C^*$ holds for every state q at the time of expansion during operation of an A^*_ϵ algorithm [70]. This results in the following corollary.

Corollary 1. *Let g be the path cost function, let h be an admissible heuristic function, and let $f = g + h$ be the cost function of* A* *operating on a state space. Further, let $\epsilon > 0$ be the parameter of the* A*-*variations* A^*_ϵ, WA*, *and* DWA*, *respectively. For $A \in \{A^*_\epsilon, \text{DWA}^*, \text{WA}^*\}$ we have: $f(q) \leqslant (1+\epsilon) \cdot C^*$ for the tentative f-values of all states at the time of expansion by A.*

Corollary 1 states a necessary condition for the expansion of a state. The result contrasts to the corresponding condition holding for the original A* algorithm. In A*, $f(q) \leqslant C^*$ holds for all states q at the time of expansion (see e.g. [40] and also the following Theorem 4). Using the unifying view it becomes possible to express the condition with the *same* cost function $f = g + h$ for all considered algorithms, even though *different* cost functions are used in DWA*, WA*, and A^*_ϵ. This in turn allows for a comparison of the respective instantiations.

As a first consequence, a potential problem becomes visible for all three methods: it may be possible that some states q satisfying the condition $C^* < f(q) \leqslant (1+\epsilon) \cdot C^*$ would be expanded by DWA*, WA*, and A^*_ϵ, but not by A*. This effect could, in some cases, even exceed the savings provided by the weighted approaches.

Now WA*, DWA*, and A^*_ϵ are compared in terms of the significance of this problem. A^*_ϵ is the algorithm in its general form, i.e. no property of a particular, second heuristic h_F can be exploited. Next, Theorem 4 considers under what conditions states are eligible for state expansion. It thereby strengthens Corollary 1, providing a formal argument in favor of WA* and DWA* over general A^*_ϵ.

Theorem 4. *Let g be the path cost function, let h be an admissible heuristic function, and let $f = g + h$ be the cost function of* A* *operating on a state space. Further, let $\epsilon > 0$ be the parameter of the* A*-*variations* A^*_ϵ, WA*, *and* DWA*, *respectively. For a snapshot of the progress of $A \in \{A^*, \text{WA}^*, \text{DWA}^*, A^*_\epsilon\}$, consider an optimal path $s, \ldots, q'$ where q' is the first state of this path that is not closed, i.e. that also appears on* OPEN*. Then*

- $A = A^*$: *$f(q) \leqslant C^*$ for all states q at the time of expansion.*
- $A = A^*_\epsilon$: *$f(q) \leqslant (1+\epsilon) \cdot C^*$ for all states q at the time of expansion.*
- $A = \text{WA}^*$: *for all states q at the time of expansion, either $f(q) \leqslant C^*$ holds or we have $f(q) > C^*$* and *$f(q) \leqslant \text{UB}$ where*

$$\text{UB} = g(q') + (1+\epsilon) \cdot \big(h(q') - h(q)\big) + h(q)$$

  That is, for $h(q)$ within the half-open interval $[0, h(q'))$, the upper bound UB *ranges from $(1+\epsilon) \cdot C^*$ to C^*, not including $(1+\epsilon) \cdot C^*$ and C^*.*
- $A = \text{DWA}^*$: *for all states q at the time of expansion, either $f(q) \leqslant C^*$ holds or we have $f(q) > C^*$* and *$f(q) \leqslant \text{UB}$ where*

$$\text{UB} = g(q') + (1+\epsilon) \cdot \left[\left(1 - \frac{d(q')}{N}\right) \cdot h(q') - \left(1 - \frac{d(q)}{N}\right) \cdot h(q)\right] + \left(1 - \frac{d(q)}{N}\right) \cdot h(q)$$

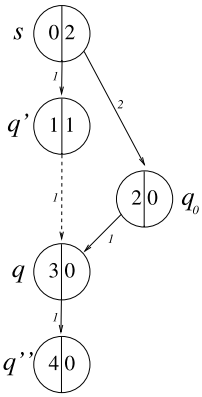
  That is, for $h(q)$ within the half-open interval $[0, \frac{N-d(q')}{N-d(q)} \cdot h(q'))$, the upper bound UB *ranges from $(1+\epsilon) \cdot C^*$ to C^*, not including $(1+\epsilon) \cdot C^*$ and C^*.*

Proof. See Appendix A. □

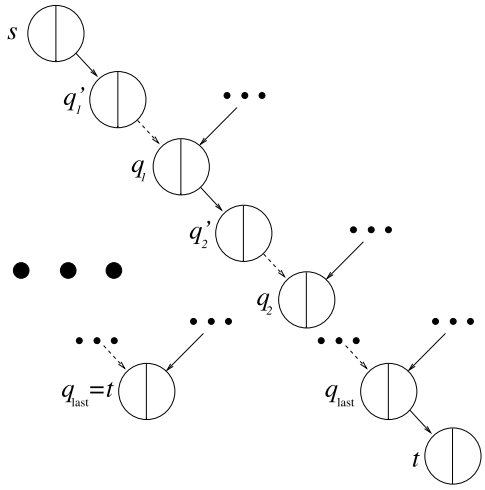
This result indicates the following: Both for WA* and for DWA*, states q with $C^* < f(q) \leqslant (1+\epsilon) \cdot C^*$ can only be eligible to expansion if their f-value also stays below the stated upper bound UB. To approach the value $(1+\epsilon) \cdot C^*$ for UB, the h-value of the eligible state must be much less than $h(q')$ and/or the eligible state must reside at a significantly deeper level in the search graph.

This contrasts to the situation in A^*_ϵ where no such additional restriction holds for the eligibility for expansion. In BDD minimization (see Section 6) as well as in many other problem domains, states with equal or similar h-values and/or depth are often expanded during a series of consecutive expansions by A* and its weighted variants. Thus, eligible states that really are far enough "below" q' (in terms of the h-value and/or depth) to be chosen for expansion, are rare. Hence, for an eligible state q, $f(q) \leqslant C^*$ is often much more typical.

Consequently it can be expected that the total number of states expanded during a run of WA* and DWA* typically is at least no more than that for A* (and often much less). If in the targeted domain A*, WA*, or DWA* have many consecutive expansions of nodes at the same or similar depth, this number is expected to still remain as low as in the situations where A^*_ϵ using $h_F = h$ or $h_F = N - d(q)$ runs into problems.



(a) Suboptimal path to an expanded state.



(b) Worst-case scenario with $\lfloor \frac{n}{2} \rfloor$ deviations, even and odd case.

Fig. 1. Examples for the behavior with and without reopening.

4. Monotonicity and reopening

In [63], Likhachev et al. provide a thorough analysis of an *Anytime Repairing* variant of A* (ARA*). In the course of their analysis, they also raise the following question: provided that weighted A* is guided by a monotone heuristic h, can states be *reopened*? And: if so, can the performance be improved by *not* reopening previously opened states? This question is of interest for all weighted variants of A*, i.e. for the algorithms $A = A^*_\epsilon$, WA*, DWA*. In the case of WA*, a concrete suggestion of Likhachev et al. was to improve the method by not reopening previously expanded states (a weighted A* algorithm with constant inflation is an integral part of their approach ARA*). Further, the authors also prove a bound on suboptimality for weighted A* with constant inflation and without reopening. It was shown that the deviation of the computed solution from the optimum cannot be greater than $1 + \epsilon$.

Following this idea of Likhachev et al., this section as well as Section 5 aims to answer the corresponding questions for *general* weighted A*. Next, an instructive example is given which shows that reopened states can exist for all choices of $A = A^*_\epsilon$, WA*, DWA*.[3]

Example 1. In Fig. 1(a), the left datum annotated at a node is the g-value, the right one is the h-value. Edges depict state transitions and the cost of the transition is annotated at each edge. The heuristic function h is monotone since Eq. (2) is respected along every path in the state space graph.

First, let $A = A^*_\epsilon$ with $h_F = h$ and let $\epsilon = \frac{1}{2}$. In the beginning, A^*_ϵ expands the initial state s with successor states q' and q_0. All states are cost minimal states on Open with $f(s) = f(q') = f(q_0) = 2$. Since q_0 has the lowest h-value, the cost of this state appears to be less estimative, i.e. the state can be expected to be the closest to a goal state. Thus q_0 is expanded next while the minimum on Open stays $f(q') = 2$. The successor state of q_0, state q, appears on Focal since

[3] The inconsistency of the cost function of WA* has already been pointed out by other authors, see e.g. [55].

$f(q) = 3 \leqslant (1 + \frac{1}{2}) \cdot 2 = 3$. Moreover, it is the state with the lowest h-value 0, and hence it is chosen as the most promising state in terms of run time. The successor state is q'' which does *not* appear on FOCAL since $f(q'') = 4 > (1 + \frac{1}{2}) \cdot 2 = 3$. As the only state left on FOCAL, q' is expanded next, *reopening* successor q. At the time of expansion of q, the best path to q via q' had not been explored yet. Hence it is $g(q) = 3 > 2 = g^*(q)$ and thus $g(q)$ now is updated to the value 2.

Next, it is easy to derive the same line of argument for the remaining choices of A, again showing that q is reopened. To see this in the case of $A = WA^*$, let $\epsilon = \frac{3}{2}$. In the case of $A = DWA^*$, let the upper bound on an optimal solution length $N = 3$ and let $\epsilon = \frac{9}{4}$.

During experiments conducted with A^*_ϵ, using the monotone heuristic from [28], this phenomenon in fact has been observed, causing high increases in run time (see Section 7). In other words, due to the relaxation, A^*_ϵ has lost much of the capability of A^* to gain from the monotonicity of h. To further analyze the operation of A^*_ϵ and its instantiations, the following new result states an upper bound for the deviation of g from g^* for an expanded state.

Lemma 1. *Let $\epsilon > 0$. The paths to expanded states found by an A^*_ϵ algorithm that is guided by a monotone heuristic may be suboptimal. However, this deviation is* bounded*, in detail*:

$$\forall q \in \text{CLOSED}:\ g(q) - g^*(q) \leqslant \epsilon \cdot \big(g^*(q) + h(q)\big) \tag{4}$$

For the special instance of Algorithm WA*, this result can be tightened to*

$$\forall q \in \text{CLOSED}:\ g(q) - g^*(q) \leqslant \epsilon \cdot k(q', q) \tag{5}$$

where q' is the first state on OPEN *on an optimal path $s, \ldots, q', \ldots, q$ at the time of expansion of q.*

Proof. See Appendix A. □

5. Preventing the reopening of states

In this section, the simple modification of weighted A* as suggested by Likhachev et al. for the case of constant inflation, is applied to A^*_ϵ and its instantiations. This yields the final methods used in the experiments (denoted using a prefix "NR" for the "Non-Reopening" variant): NRA^*_ϵ as well as two instantiations of NRA^*_ϵ, called NRWA* and NRDWA*. Notice that NRWA* is not really a new method as it has been proposed and implemented by Likhachev et al. [63] (however, in a different context).

Consider the following change of operation for A^*_ϵ, if the method finds a better path for a closed state q, *this better path is ignored*, i.e. $g(q)$ is *not* updated. Otherwise, method NRA^*_ϵ follows the usual operation of A^*_ϵ.

As has been said in the previous section, we are interested in a general bound for suboptimality, i.e. a bound for the framework A^*_ϵ under the above modification (denoted NRA^*_ϵ). Although ϵ-admissibility cannot be guaranteed for NRA^*_ϵ in general, still the following result can be shown:

Theorem 5. *Let N be an upper bound on an optimal solution length. When driven by a monotone heuristic, Algorithm* NRA^*_ϵ *always finds a solution not exceeding the optimal cost by a factor greater than $(1 + \epsilon)^{\lfloor \frac{N}{2} \rfloor}$.*

Proof. See Appendix A. □

Basically, the proof follows a similar line of argument as the proof for ϵ-admissibi-lity of A^*_ϵ [70], except that, due to the modified behavior of the algorithm, one has to account for the following consequence. In NRA^*_ϵ, the g-value of states on an optimal path may irrecoverably be affected by deviations from the optimum, g^*. By Lemma 1, states might be expanded while the best known path to them is still suboptimal. Due to the modified behavior of NRA^*_ϵ, no reopening/improvement can take place later. This effect increases the maximum deviation on an optimal path. The extent of the deviation is determined in the worst-case scenario. Let N be the maximal length of an optimal path. Since *two* nodes must always be involved for a deviation of a g-value to occur (see the proof of Lemma 1), the deviation of a g-value from g^* increases at most $\lfloor \frac{N}{2} \rfloor$ times. In Fig. 1(b), dashed transition are "late" transitions, i.e. the state they lead to has already been opened along a suboptimal path different from p. State q_{last} is the last state that has been prematurely opened along such a sideway and, thus, is affected by a deviation of the g-value. We have $q_{\text{last}} = q_{\lfloor \frac{N}{2} \rfloor}$, regardless of whether N is odd or even (see Fig. 1(b)). The proof then is an induction on $i = 1, \ldots, \lfloor \frac{N}{2} \rfloor$.

Notice that the result for the case $\epsilon = 0$ states that in this case operation of NRA^*_ϵ coincides with that of A^* when driven by a monotone heuristic. This is expected since $A^*_{\epsilon=0} = A^*$. Moreover, by Theorem 1, A^* always finds optimal paths to expanded states. Consequently the described modification in the behavior of $A^*_{\epsilon=0}$ does not affect operation.

The fact of being bounded suboptimality is not only present for the case of constant inflation but also for general A^*_ϵ, is a new (theoretical) result in itself. However, this general bound is exponential in the depth of the search. In practice, it can

still be useful for smaller values of ϵ and N, as it is the case for such examples as approximate BDD minimization. Notice that the construction of the worst-case already gives a strong intuition that this would be a rare event. As far as the domain of approximate BDD minimization is concerned, this expectation has been strengthened by the performance of NRA$^*_\epsilon$ which is far off the worst-case (see Section 7). Reconsidering the upper bounds stated in Theorem 4, it is straightforward to derive the following corollary:

Corollary 2. *Let g be the path cost function, let h be a monotone heuristic function, and let $f = g + h$ the cost function of* A* *operating on a state space. Further, let N be an upper bound on an optimal solution length and let $\epsilon > 0$ be the parameter of the* A*-*variations* NRA$^*_\epsilon$, NRWA*, *and* NRDWA*, *respectively. Then*

- A = NRA$^*_\epsilon$: $f(q) \leqslant (1+\epsilon)^{\lfloor \frac{N}{2} \rfloor} \cdot C^*$ *for all states q at the time of expansion, and*
- A $\in$ {NRWA*, NRDWA*}: $f(q) \leqslant (1+\epsilon) \cdot C^*$ *for all states q at the time of expansion.*

The proof is analogous to the one of Theorem 4. The only differences between the two proofs are in the places where, in the proof of Theorem 4, the g-value of the first open state on an optimal path could be bounded by its respective g^*-value. Due to the modification of A$^*_\epsilon$ in the revised version NRA$^*_\epsilon$, this is not a valid conclusion anymore, i.e. the costs of states on optimal paths can be higher than the optimal costs (however, they are still provably bounded, see Lemma 1 as well as the proof of Theorem 5). For more details, see the proof of Theorem 4.

The result states that the respective NRA$^*_\epsilon$-versions may expand more states than the original approaches, i.e. the obtained upper bounds for the f-value of states are larger. The reason is that, during derivation of the bounds, we must account for a possible degradation of the g-value of nodes on the optimal path. As follows from the discussion of the worst-case of NRA$^*_\epsilon$, there is sufficient evidence that this will be a rare event for a number of application domains. Despite the result, for the appropriate domains an increase in run time as caused by a larger number of state expansions is not expected in practice. It can be assumed that the revised algorithms will still often behave as forecasted by Theorem 4. This is confirmed later by experiments for the BDD domain in Section 7.

6. Approximate BDD minimization – a problem domain for weighted A*

Reduced ordered *Binary Decision Diagrams* (BDDs) were introduced in [9]. There are many fields of application for BDDs in AI, including software model checking [29], sparse-memory applications [44,89], BDD-based planning [2,3,17,18,34,45] and symbolic (BDD-based) heuristic search [39,73]. BDDs often enhance classical search methods of AI, such as the A* algorithm [30,43]. Moreover, Reffel and Edelkamp also used a BDD-based version of A* to enhance model checking for hardware verification [75]. This work also is an example of research at the intersections of AI and VLSI CAD. In VLSI CAD, BDDs are well known from hardware verification and logic synthesis.

BDD is a *graph-based* data structure for the representation of Boolean functions. Redundant nodes in the graph, i.e. nodes not needed to represent a Boolean function f, can be eliminated. BDDs allow a unique (i.e. canonical) representation of Boolean functions. At the same time they allow for a good trade-off between efficiency of manipulation and compactness of the representation.

Exact BDD minimization requires to find the optimal variable ordering which yields the minimum BDD size (i.e. the minimum number of nodes). It is known to be an NP-complete problem [5]. Approaches for exact minimization have been proposed in the hardware community. The algorithm of Friedman and Supowit [36] works on truth table representations and has an exponential time complexity of $O(n^2 \cdot 3^n)$. This was later reduced to $O(n \cdot 3^n)$ by application of a 2-phase bucket-sort technique by Sieling and Wegener [82]. Other approaches aimed more at improvement of performance without reducing the worst-case complexity [24,27,42,46]. The most recent suggestion is to use the A* algorithm [28]. Besides the fact that they shed additional light on the problem, there are strong reasons for this research: in several real-world applications in VLSI CAD, BDDs are directly mapped to circuits. Henceforth, the BDD size can be directly transferred to the resulting chip area and it is a significant drawback to start from a BDD whose size has been optimized by the use of heuristics. This particularly holds when methods like the well-known sifting algorithm [25,26,76] which are based on a hill-climbing framework are far away from the optimum. In the experiments described in this paper, the percentage loss in quality for sifting was up to more than 80% (see Section 7.3). In [84], an instructive example is given where sifting yields a BDD as large as twice the size of the optimum. Moreover, BDD sizes that result from sifting have been compared to known upper bounds on the optimal sizes which have been obtained by the use of simulation-based approaches like SA or EAs [4,23]. This revealed that the difference can be as large as orders of magnitudes.

In the past, numerous research papers have addressed BDD-based approaches for the automated design or logic optimization of FPGAs (e.g. [14–16,32,68]), of Pass Transistor Logic (PTL) or other multiplexor-based design styles [10,35,66,67]. Of note is that the academic research has been strongly supported by the release of publicly available BDD-based design automation tools like the logic optimization system BDS [86] and the PTL-oriented synthesizing tool PTLS [80].

In this paper, the problem domain of approximate BDD minimization is added to the range of more typical AI problem domains such as the sliding-tile puzzle or planning problems like logistics or blocksworld problems. To keep the paper self-contained, this section provides the necessary formal background of Boolean functions and BDDs.

6.1. Boolean functions

The following is an introduction of the notations for Boolean functions used throughout the paper.

Definition 2. Let $\mathbf{B} = \{0, 1\}$ and $X_n = \{x_1, \ldots, x_n\}$ where $x_1, \ldots, x_n$ are Boolean variables. Let $m, n \in \mathbb{N}$. A mapping

$$f : \mathbf{B}^n \to \mathbf{B}^m$$

is called a *Boolean function*. In the case of $m = 1$ we say f is a *single-output function*, otherwise f is called a *multi-output function*. To put emphasis on the arity n of f, we may choose to write $f^{(n)}$ instead of f. A multi-output function $f : \mathbf{B}^n \to \mathbf{B}^m$ can be interpreted as a family of m single-output functions $(f_i^{(n)})_{1 \leqslant i \leqslant m}$.

To achieve a standard, the set of variables of a Boolean function $f^{(n)}$ will always be assumed to be X_n. If not stated otherwise, Boolean functions are assumed to be *total* (*completely specified*), i.e. there exists a defined function value for every vector of input variables. The Boolean functions constantly mapping every variable to 1 (to 0) are denoted *one* (*zero*), i.e.

$$\text{one} : \mathbf{B}^n \to \mathbf{B}^m;\ (x_1, x_2, \ldots, x_n) \mapsto 1$$
$$\text{zero} : \mathbf{B}^n \to \mathbf{B}^m;\ (x_1, x_2, \ldots, x_n) \mapsto 0$$

An interesting class of Boolean functions are (partially) symmetric functions. Later, in Section 6.4, it is explained how the A*-based approach to exact BDD minimization exploits (partial) symmetry to reduce run time.

Definition 3. Let $f : \mathbf{B}^n \to \mathbf{B}^m$ be a multi-output function. Two variables x_i and x_j are called *symmetric*, iff

$$f(x_1, \ldots, x_i, \ldots, x_j, \ldots, x_n) = f(x_1, \ldots, x_{i-1}, x_j, x_{i+1}, \ldots, x_{j-1}, x_i, x_{j+1}, \ldots, x_n)$$

Symmetry is an equivalence relation which partitions the set X_n into disjoint classes $S_1, \ldots, S_k$ called the *symmetry sets*. They are sets of variables which are pairwise symmetric. A function f is called *partially symmetric*, iff it has at least one symmetry set S with $|S| > 1$. If a function f has only one symmetry set $S = X_n$, then it is called *totally symmetric*.

6.2. Shannon decomposition

The well-known theorem [79] allows decomposing Boolean functions into "simpler" sub-functions. In the following definition, a cofactor of a Boolean function f is defined as the function derived from f by fixing a variable of f to a value in $\mathbf{B}$.

Definition 4. Let $f : \mathbf{B}^n \to \mathbf{B}^m$ be a Boolean function. The *cofactor* of f with respect to $x_i = c$ ($c \in \mathbf{B}$) is the function $f_{x_i=c} : \mathbf{B}^n \to \mathbf{B}^m$. For all variables in X_n it is defined as

$$f_{x_i=c}(x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) = f(x_1, x_2, \ldots, x_{i-1}, c, x_{i+1}, \ldots, x_n)$$

Repeated cofactoring yields cofactors with respect to more than one variable, e.g. for $1 \leqslant k \leqslant n$ and a set of k variable indices $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$, for $\{x_{i_1}, \ldots, x_{i_k}\} \subseteq X_n$ and $(c_1, \ldots, c_k) \in \mathbf{B}^k$, the function $f_{x_{i_1}=c_1, x_{i_2}=c_2, \ldots, x_{i_k}=c_k}$ is a cofactor in multiple variables. This cofactor is equivalent to

$$(f_{x_{i_1}=c_1, x_{i_2}=c_2, \ldots, x_{i_{j-1}}=c_{j-1}, x_{i_{j+1}}=c_{j+1}, \ldots, x_{i_k}=c_k})_{x_{i_j}=c_j}$$

for any $1 \leqslant j \leqslant k$. Let $f : \mathbf{B}^n \to \mathbf{B}^m$ be a Boolean function and let $x_i \in X_n$. Then function f is said to *depend essentially* on x_i iff

$$f_{x_i=0} \neq f_{x_i=1}$$

Theorem 6. *Let $f : \mathbf{B}^n \to \mathbf{B}^m$ be a Boolean function (over X_n). For all $x_i \in X_n$ we have*:

$$f = x_i \cdot f_{x_i=1} + \bar{x}_i \cdot f_{x_i=0} \tag{6}$$

6.3. BDDs

In the following, a formal definition of BDDs is given. We start with purely *syntactical* definitions by means of *Directed Acyclic Graphs* (DAGs). First, single-rooted *Ordered Binary Decision Diagrams* (OBDDs) are defined. This definition is extended to multi-rooted graphs, yielding *Shared* OBDDs (SBDDs). Next, the *semantics* of SBDDs is defined, clarifying how Boolean functions are represented by SBDDs. After that, reduction operations on SBDDs are introduced which preserve the semantics of an SBDD. This leads to the final definition of reduced SBDDs that will be called BDDs for short.

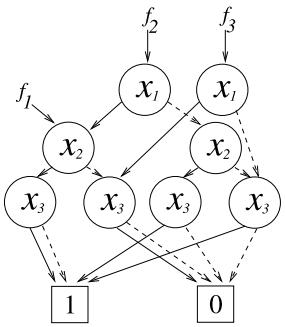


Fig. 2. A shared OBDD (SBDD).

6.3.1. Syntactical definition of BDDs

Definition 5. An *Ordered Binary Decision Diagram* (OBDD) is a pair (π, G) where π denotes the variable ordering of the OBDD and G is a finite DAG $G = (V, E)$ (V denotes the set of *vertices* and E denotes the set of *edges* of the DAG) with exactly one root node (denoted *root*) and the following properties:

- A node in V is either a non-terminal node or one of the two terminal nodes in $\{\mathbf{1}, \mathbf{0}\}$.
- Each non-terminal node v is labeled with a variable in X_n, denoted $\text{var}(v)$, and has exactly two child nodes in V which are denoted $\text{then}(v)$ and $\text{else}(v)$.
- On each path from the root node to a terminal node the variables are encountered at most once and in the same order. More precisely, the variable ordering π of an OBDD is a bijection

  $$\pi : \{1, 2, \ldots, n\} \to X_n$$

  where $\pi(i)$ denotes the ith variable in the ordering. The above condition "in the same order" states that for any non-terminal node v we have

  $$\pi^{-1}\big(\text{var}(v)\big) < \pi^{-1}\big(\text{var}\big(\text{then}(v)\big)\big)$$

  iff $\text{then}(v)$ is also a non-terminal node and

  $$\pi^{-1}\big(\text{var}(v)\big) < \pi^{-1}\big(\text{var}\big(\text{else}(v)\big)\big)$$

  iff $\text{else}(v)$ is also a non-terminal node.

For convenience, variable orderings will be given as sequences of variables, e.g. we write x_3, x_1, x_2 to express that $\pi^{-1}(x_3) = 1 < \pi^{-1}(x_1) = 2 < \pi^{-1}(x_2) = 3$.

Definition 6. A *Shared* OBDD (SBDD) is a tuple (π, G, O). G is a rooted, possibly multi-rooted DAG (V, E) which consists of a finite number of graph components. These components are OBDDs, all of them respecting the same variable ordering π. $O \subseteq V \setminus \{\mathbf{1}, \mathbf{0}\}$ is a finite set of output nodes $O = \{o_1, o_2, \ldots, o_m\}$. An SBDD has the following properties:

- A node in V is either a non-terminal node or one of the two terminal nodes in $\{\mathbf{1}, \mathbf{0}\}$.
- Every root node of the component OBDD graphs must be contained in O (but not necessarily vice versa).

Example 2. An example of an SBDD is given in Fig. 2. Solid lines are used for the edges from v to $\text{then}(v)$ whereas dashed lines indicate an edge between v and $\text{else}(v)$. The nodes pointed to by f_1, f_2 and f_3 are output nodes. Notice that every root node is an output node (pointed to by f_2 and f_3) and that not every output node is a root node (see the node pointed to by f_1).

Also note that in SBDDs multiple graphs can share the same node, a property which helps to save nodes and to reduce the size of the diagram. The idea behind the set O is to declare additional non-terminal, non-root nodes as nodes representing Boolean functions. This will be clarified in the next section when the semantics of BDDs is defined. Notice that SBDDs as well as OBDDs have at most two terminal nodes which are shared by the components.

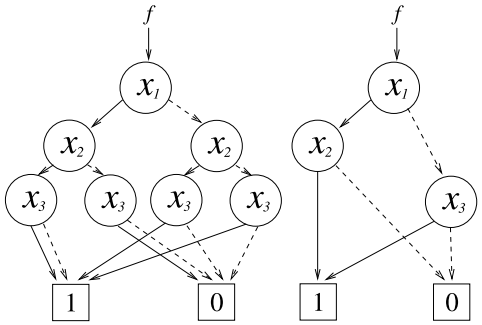


Fig. 3. Two different SBDDs for $f : (x_1, x_2, x_3) \mapsto x_1 \cdot x_2 + \bar{x}_1 \cdot x_3$.

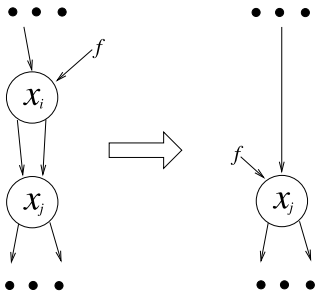


Fig. 4. Deletion Rule for SBDD-reduction.

6.3.2. Semantical definition of BDDs

Definition 7. An SBDD $(\ldots, G, O)$, over X_n with $O = \{o_1, o_2, \ldots, o_m\}$ represents the multi-output function $f := (f_i^{(n)})_{1 \leqslant i \leqslant m}$ defined as follows:

- If v is the terminal node **1**, then $f_v = \text{one}$, if v is the terminal node **0**, then $f_v = \text{zero}$.
- If v is a non-terminal node and $\text{var}(v) = x_i$, then f_v is the function

$$f_v(x_1, \ldots, x_n) = x_i \cdot f_{\text{then}(v)}(x_1, \ldots, x_n) + \bar{x}_i \cdot f_{\text{else}(v)}(x_1, \ldots, x_n)$$

- For $1 \leqslant i \leqslant m$, f_i is the function represented by the node o_i.

The expression $f_{\text{then}(v)}$ ($f_{\text{else}(v)}$) denotes the function represented by the child nodes then(v) (else(v)). At each node of the SBDD, essentially a Shannon decomposition (see Theorem 6) is performed. In this, an SBDD recursively splits a function into simpler sub-functions.

Two different variable orderings yield two different BDDs (see e.g. Fig. 7). Even if the considered variable ordering is fixed, still there exist several possibilities of representing a given function: in Fig. 3 we see two different SBDDs respecting the same variable ordering x_1, x_2, x_3, representing the same function $f : \mathbf{B}^3 \to \mathbf{B}$; $(x_1, x_2, x_3) \mapsto x_1 \cdot x_2 + \bar{x}_1 \cdot x_3$. Figs. 4 and 5 illustrate reduction operations on SBDDs which transform an SBDD into an irreducible form, while the function represented by the SBDD is preserved. With the Deletion Rule, redundant nodes are deleted. Subsequent application of the Merging Rule identifies isomorphic sub-graphs. This leads to an SBDD respecting a given variable ordering which is unique up to graph isomorphism.

A *reduced* SBDD then is the final form of binary decision diagrams called BDD.

Definition 8. An SBDD is called *reduced* iff there is no node where one of the reduction rules (i.e., the Deletion or the Merging Rule) applies.

The following theorem [9] holds:

Theorem 7. *BDDs are a canonical representation of Boolean functions, i.e. the BDD-representation of a given Boolean function with respect to a fixed variable ordering is unique up to isomorphism.*

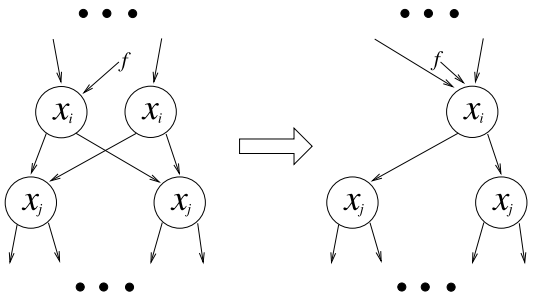


Fig. 5. Merging Rule for SBDD-reduction.

6.4. Finding a best BDD variable ordering by path cost minimization

6.4.1. Idea

In this paper, approximate BDD minimization is achieved by weighted A*. This approach is based on a previous work [28] which describes exact BDD minimization as a problem of finding a minimum cost path that is solved by A*. To keep the paper self-contained, the basic concept of this work is briefly reviewed in this section.

The problem of exact BDD minimization is the problem of finding an optimal variable ordering, i.e. one that leads to a minimum number of BDD nodes. In [28], the problem of finding an optimal variable ordering is expressed as the problem of finding a minimum cost path from the initial state $\emptyset$ to the goal state X_n in the state space 2^{X_n}.

Sets of variables $q \subseteq X_n$ are successively growing from $\emptyset$ to X_n: q is extended at each transition by a variable $x_i \in X_n \setminus q$, i.e. $q \xrightarrow{x_i} q \cup \{x_i\}$. The algorithm starts in the initial state $\emptyset$ and progresses until the goal state X_n is reached. As described before in Section 2.1, A* finds a path $p^*(X_n)$ from $\emptyset$ to X_n with minimal cost. The optimal path $p^*(X_n)$ is an optimal sequence of transitions. Consequently, there must exist a permutation σ of the numbers $1, \ldots, n$ (i.e., $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ is a bijection) such that the aforementioned minimal cost is the accumulated transition cost for the transitions

$$\emptyset \xrightarrow{x_{\sigma(1)}} \{x_{\sigma(1)}\} \xrightarrow{x_{\sigma(2)}} \{x_{\sigma(1)}, x_{\sigma(2)}\} \xrightarrow{x_{\sigma(3)}} \cdots \xrightarrow{x_{\sigma(n)}} X_n$$

along $p^*(X_n)$. The sequence of variables occurring on this path obviously defines a variable ordering.

The basic idea of the approach is the following: the above ordering annotated along the minimum cost path is intended to be optimal. This means, $f^*(X_n)$ is intended to be the number of nodes in the BDD with the ordering $x_{\sigma(1)}, x_{\sigma(2)}, \ldots, x_{\sigma(n)}$. Given, that this already holds, the sequence of variables along $p^*(X_n)$ must be an optimal variable ordering, yielding the minimum BDD size.

To achieve this, an appropriate cost function is chosen. A sequence of variables occurring along the transitions from $\emptyset$ to a non-goal state has the semantics of a *prefix* of a variable ordering. That is, a path of length k defines the positions of the first k variables in a variable ordering. The key idea of [28] is to define the cost function such that the number of nodes in the first k levels of a BDD is taken as the cost of the corresponding path of length k. In this, the method does not perform variable transpositions (as in local search approaches) but incrementally generates the ordering by adding one variable after the other.

6.4.2. Example

An example of a run of the A*-based approach of [28] is given in Fig. 6. The algorithm is applied to the initial BDD in Fig. 7(a), which represents the function

$$f : \mathbf{B}^4 \to \mathbf{B};\ (x_0, x_1, y_0, y_1) \mapsto x_0 \cdot x_1 + y_0 \cdot y_1$$

First, this function is partially symmetric with two symmetry sets (see Definition 3). The first set is $\{x_0, x_1\}$, the second is $\{y_0, y_1\}$. Because of the symmetry, for any two variables within one set, the structure of a representing BDD is preserved if their positions in a variable ordering are swapped (the only change is a renaming of the respective node labels). Second, the function is a four-input instance of the n-input "Achilles heel" function (n has to be even) given in [9]. The BDD of this more general function has only linear size for any ordering that respects an (L, R)-partition of the variables where L, R are the two symmetry sets. However, for an interleaved ordering where variables from different sets are neighbored, the BDD is of exponential size (for more details, see [9]). This demonstrates an "Achilles heel" of BDDs, i.e. their crucial dependency on the ordering.

All BDDs depicted in Fig. 7 are BDDs which actually have been built during the run. The graphical outputs have been generated by use of the GraphViz-interface of CUDD [31,83]. Different from the previous illustrations, a distinct identifier is annotated to each node whereas the actual node label (i.e., the respective variable) is annotated at the BDD levels (due to the ordering restriction in Definition 5, the variable label is the same for all nodes of a BDD level). This enables references to particular nodes in the BDD in the textual descriptions of the figures. Notice that the BDDs of Fig. 7 also have some

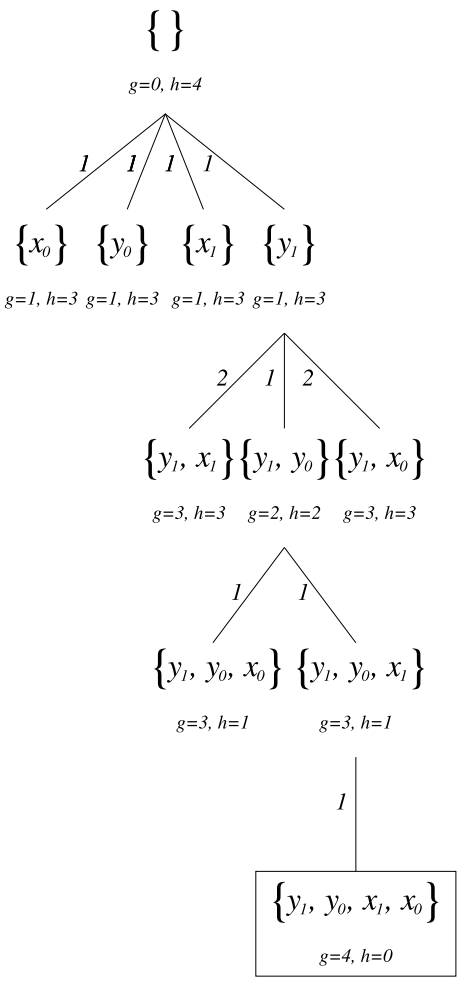


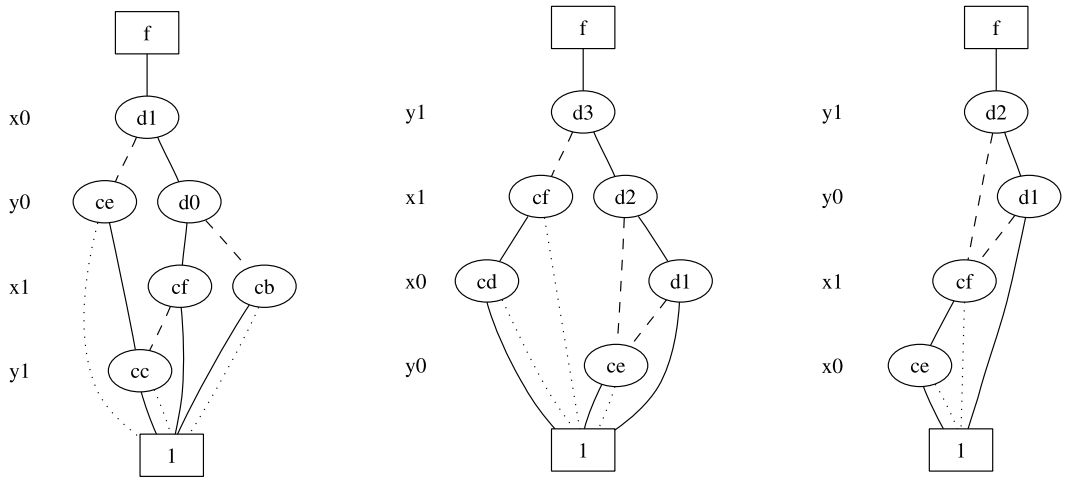
Fig. 6. A* applied to a BDD for a four-input "Achilles heel" function with a bad initial ordering.

dotted edges besides the dashed and solid ones. These indicate the use of *Complemented Edges* (CEs). CEs are an important extension of the basic BDD concept and have been described in [1,8]. A CE is an ordinary edge that is tagged with an extra attribute, the *complement bit*. This bit is set to indicate that the connected sub-graph must be interpreted as the complement of the formula that it represents. CEs allow to represent both a function and its complement by the same node, modifying the edge pointing to that node instead. As a consequence, only one constant node is needed. Usually the node **1** is kept, allowing the function zero to be represented by a CE to **1**.

The initial BDD respects the interleaved ordering x_0, y_0, x_1, y_1, which is a suboptimal ordering. In Fig. 6, states are sets of variables which constitute the nodes of the search graph. The g-value and the h-value are annotated at each state. Edges depict state transitions (which are always from the top to the bottom). The transition costs (edge costs) are annotated at the edges. A detailed description of the heuristic function h used follows later in Section 6.4.3.

The initial state is the empty set which is expanded to the four successors $\{x_0\}, \{y_0\}, \{x_1\}, \{y_1\}$. The edges leading to them all have costs of 1 because for every successor one root node is established at the first BDD level. Since g- and h-values are identical for the first four open nodes, a second-order tie-breaking rule (which, in this case, is motivated by efficiency aspects) selects the state $\{y_1\}$. During the next steps, ties in the value $f = g + h$ are resolved (by the first-order tie-breaking rule) in favor of the state with the lower h-value where possible.

The expansion of state $\{y_1\}$ generates the successors $\{y_1, x_1\}$, $\{y_1, y_0\}$, and $\{y_1, x_0\}$. The order of elements in the set notation gives the path taken from $\emptyset$ to the state (this saves space in the illustration). In a formal sense, paths are ordered sequences and states are unordered sets (and in particular, the distinct paths (y_1, y_0) and (y_0, y_1) would both end in state $\{y_1, y_0\}$). However, in this small example, a state where a second, cheaper path is found, does not occur and hence the (only) path to a state is simply given by the order of elements in the state sets. The successor state $\{y_1, x_1\}$ has a g-value of three. This reflects the total of three nodes in the first two levels of a BDD for the example function given that variable y_1 is situated at the root and variable x_1 resides at the second level (see 7(b)). Notice that the structure of the first two (or, in general: k) levels holds regardless of the variable ordering in the part of the BDD below the second (or, in general: kth)



(a) Initial ordering. (b) Intermediate ordering. (c) Optimal ordering.

Fig. 7. BDDs for initial, an intermediate, and the optimal ordering.

level. This follows from a well-known theorem of Friedman and Supowit (for more details and a rigorous proof see [36]). This is important because otherwise the path cost function g which maps states to costs of the currently best known path to them would have to map a state to more than one value (i.e., it would not be a well-defined function).

Due to the partial symmetry of the example function, the BDDs with an ordering $y_1, x_0, \ldots$ have the same g-value of three as the BDD depicted in Fig. 7(b).

In the next step, state $\{y_1, y_0\}$ is expanded since it has the lowest h-value in the set of open states with minimal f-value of four. Again, for reasons of partial symmetry, the BDDs with the orderings $y_1, y_0, x_0, \ldots$ and $y_1, y_0, x_1, \ldots$ have identical g-value of three (and the same h-value of one). From the set of open states with the minimal f-value four, the two states $\{y_1, y_0, x_0\}$ and $\{y_1, y_0, x_1\}$ have the lowest h-value and therefore are selected by the first-order tie-breaking rule. The second-order tie-breaking rule then selects the state $\{y_1, y_0, x_1\}$ for expansion. This results in the optimal ordering y_1, y_0, x_1, x_0 and a BDD with only four nodes (see 7(c)). Notice that this has been achieved with only four state expansions (i.e., the number of expansions is linear in the number of input variables n, which corresponds to the problem size).

For brevity, several details and optimizations of the method described in [28] have been omitted. E.g., the algorithm is able to detect the symmetry sets of the variables. When operating with the according option switch USE_SYMMETRY, only the first variable of a symmetry set is processed (to obtain a simple, instructive example, this optimization has not been used in the example run depicted in Fig. 6). The other variables can be skipped since it is clear that the resulting g- and h-values must be the same as for the first variable (e.g. see the g- and h-values of the states $\{y_1, x_1\}$ and $\{y_1, x_0\}$). Henceforth, if symmetry is exploited, also the number of generated states stays linear in the problem size n.

Notice that, for the calculation of the g- and h-values, the BDD that is subjected to optimization is used itself. For this purpose, the BDD often has to be reordered with respect to the state that is currently processed. The basic building-block for reordering is a swap of two adjacent BDD levels. This can be done with a graph operation that only needs to touch the vertices of the two affected BDD levels. Nevertheless this operation is time-consuming if the two BDD levels are large. Since this operation is needed with every expansion, the approach is often more time-bounded than memory-bounded. This is different from the application of A* in classical AI domains where the time needed for an expansion is often constant, and thus in practice these approaches are more memory-bounded. Several optimizations in [28] address the efficiency of the dynamical reordering needed at a state expansion.

6.4.3. Heuristic function

Next, the heuristic function used in [28] is briefly reviewed. It was derived using a relaxation of the original model and problem: consider a BDD representing f with an ordering which first $|q|$ variables constitute q.

The *original* problem is to determine the minimum number of nodes in the remaining lower part of the BDD such that f remains correctly represented; or, equivalently, such that all cofactors of f with respect to all variables in q remain correctly represented.

The *relaxed* problem then is to determine the number of such distinct cofactors. This is a lower bound on the minimum number of nodes to represent them. To see this, notice that at least one additional node is needed for every distinct cofactor: this node is the root node of the subgraph representing the cofactor. This cannot be said for the nodes different from this root since the subgraphs representing the cofactors may share nodes: e.g., in Fig. 7(b), the BDD node labeled “ce” is shared by the subgraphs rooted at the nodes labeled “d1” and “d2”.

More precisely, let $f: \mathbf{B}^n \to \mathbf{B}^m$; $f = (f_i^{(n)})_{1 \leqslant i \leqslant m}$ be a Boolean multi-output function that depends essentially on all its input variables. For $1 \leqslant k \leqslant n$, let $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$ be a set of k variable indices. Consider a state $q \in 2^{X_n}$ with k elements that correspond to the aforementioned set of k indices, i.e. $q = \{x_{i_1}, \ldots, x_{i_k}\} \subseteq X_n$. For q, let

$$\operatorname{cof}(f, q) = \left\{ (f_i)_{x_{i_1}=a_1,\ldots,x_{i_k}=a_k} \text{ non-constant } \mid 1 \leqslant i \leqslant m, (a_1, \ldots, a_k) \in \{0,1\}^k \right\} \tag{7}$$

For state q, $\operatorname{cof}(f, q)$ counts the number of cofactors with respect to the variables in q. E.g., in Fig. 7(b), the cofactors with respect to the variables in $\{y_1, x_1\}$ are represented by the BDD nodes labeled "cd", "d1", and "ce". More formally, the set $\operatorname{cof}(f, q)$ is the set of all distinct, non-constant (single-output) cofactors of f (f is interpreted as a family of n-ary single-output functions) with respect to all variables in q. The heuristic function $h: 2^{X_n} \to \mathbb{N}$ used in the approach is

$$h(q) = \max\left(\left|\operatorname{cof}(f, q)\right|, n - |q|\right) \tag{8}$$

It is straightforward to see that $h(q)$ is a lower bound on the minimum number of BDD nodes in the levels $|q| + 1, \ldots, n$ of any BDD

- that represents f, and
- where the variables in q are situated at the first $|q|$ levels.

First, all distinct cofactors in $\operatorname{cof}(f, q)$ must be represented by different nodes in levels $|q| + 1, \ldots, n$, since otherwise the BDD would not represent f. Second, in every level from $|q| + 1$ to n there is at least one node since f is assumed to depend essentially on all input variables. This yields the second term $n - |q|$. Hence, the maximum of both lower bounds, i.e. $h(q)$, must also be a lower bound.

E.g. for the state in Fig. 7(b), the resulting h-value is

$$\begin{aligned} h(\{y_1, x_1\}) &= \max\left(\left|\operatorname{cof}(f, \{y_1, x_1\})\right|, n - \left|\{y_1, x_1\}\right|\right) \\ &= \max(3, 4 - 2) \\ &= 3 \end{aligned}$$

It can be computed effectively with a top down graph traversal on the BDD, counting the number of direct references from the upper nodes to the nodes in the lower part of the BDD [24]. Of note is that this heuristic function has the convenient property of *monotonicity* (see Definition 1). For more details and a rigorous proof, see [28].

7. Experimental results

To evaluate the algorithms in discussion, the respective methods have been applied to several problem domains. The first suite of experiments tackles the problem of approximate BDD minimization as defined in Section 6.4. For this purpose, the respective methods have been implemented by the authors, starting from the A*-based approach of [28] which combines A* with Branch and Bound (B&B). For a comparison of plain concepts, we did not use a combination of A* and B&B here. All algorithms make use of Dial/Johnson queues to quickly determine the respective next minimum from OPEN [21,47]. To put up a testing environment, all algorithms have been integrated into the CUDD package [83]. By this it is guaranteed that they run in the same system environment. The present expertise in this particular domain also allowed for the inclusion of domain-dependent approaches.

In a second suite, weighted A* is applied to STRIPS benchmark problems in typical planning domains. They have been obtained from previous planning competitions such as AIPS2000, IPC3 and IPC4, and from distributions of problem solvers that have been made available to the public, such as the domain-independent planner HSP2 by Bonet and Geffner [6,7].

All experimental results have been carried out on a machine with a Xeon processor running at 3.2 GHz with 12 GB of memory. For the BDD domain, a run time limit of 3600 CPU seconds was imposed. Within the given time limit, a total of 28 benchmark functions from LGSynth93 [19], a benchmark suite of combinational and sequential circuits could be minimized with A* and its variants. For this purpose, the logic-level description given in a *Berkeley Logic Interchange Format* (BLIF) file has been used to build the representing BDDs. Since the BDD domain is more time-bounded than other classical AI domains (see Section 6.4), the memory requirement of all evaluated methods never exceeded 500 MB, hence no memory limit was applied.

For the planning domains, the performance of A*, WA* and NRWA* for different weights was compared by means of the domain-independent planner HSP2 and our implementation of the non-reopening variant NRWA* that has been obtained by appropriate modifications of the source code of HSP2. For this purpose, those STRIPS problems in the aforementioned planning domains have been chosen that we succeeded to solve optimally by A* on our machine within 100 CPU seconds and two GB of memory. The number of these problems varies, starting from 38 and 23 problems in the PSR and Blocksworld domain, respectively, down to only two problems in the Satellite domain.

Additionally, in Section 7.3, also hard instances of the BDD domain and the STRIPS domains (i.e., problems that cannot be solved with A* within the given time and memory limit) have been solved by weighted A*.

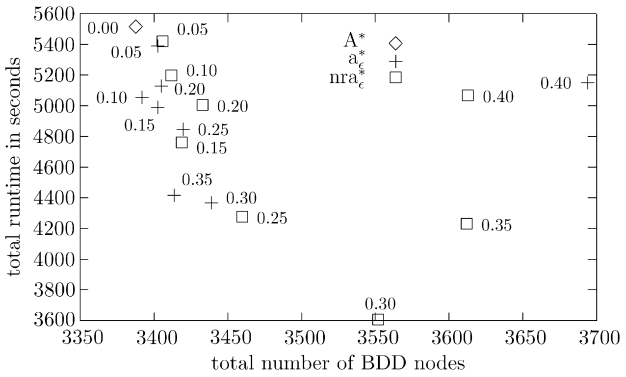


Fig. 8. Trading off run time for solution quality with a^*_ϵ and nra^*_ϵ.

In all experiments run time, solution length and the number of generated and expanded search graph nodes have been acquired, respectively.

7.1. Application to approximate BDD minimization

7.1.1. Aim and methodology

Two previous methods have been implemented: the first is called a^*_ϵ which instantiates A^*_ϵ (see [70] and also Section 2.3.3) in the BDD context. As the focal heuristic of a^*_ϵ, $h_F(q) = N - d(q)$ has been chosen, where $d(q)$ denotes the depth of state q in the search graph and N is an upper bound on an optimal solution length (see also Section 2.3.3). The second is DWA* (see [72] and also Section 2.3.2). In the following, the parameter ϵ of the weighting will be referred to as the "degree of relaxation". Besides the approaches WA* and the non-reopening variant NRWA* (see Section 2.3.1), new, non-reopening variants of two known methods have been implemented as the corresponding methods nra^*_ϵ (non-reopening variant of a^*_ϵ) and NRDWA* (non-reopening version of DWA*). The aim of the experiments was the analysis and comparison of the respective different methods' suitability to let the user trade solution quality for run time. The user controls the degree of relaxation, resulting in different run times and final BDD sizes.

Figs. 8, 10, and 13–15 depict the mean values of the two dimensions run time and quality for every group that is constituted by a different degree of relaxation. In the graphs, the degree used is annotated at the points as the applied value of ϵ. The progress of the gain in run time and the loss in quality, with an increasing degree of relaxation ϵ is illustrated in Figs. 11 and 12: Here, mean values of the gain and loss in every group are depicted.

7.1.2. Comparison of a^*_ϵ to nra^*_ϵ

In a first series of experiments, A*, a^*_ϵ, and nra^*_ϵ have been applied to the benchmark circuits of the test suite. Fig. 8 depicts points on the space spanned by the two dimensions solution quality (i.e. the number of BDD nodes) and total run time for the whole test suite (in CPU seconds). Both methods show a significant gain in run time when compared to A*: at a degree of relaxation of 30%, a^*_ϵ achieves a reduction in run time of 20.7% when compared to A*. For nra^*_ϵ, the gain is 34.6%. The results also show that the run time for both methods is not always monotonic decreasing when allowing for a rising degradation of solution quality. Instead a^*_ϵ tends to jump across the spanned space when applying small increases in relaxation. This limits the usefulness of a^*_ϵ for the desired run time/quality tradeoffs. In contrast, the plot resulting from the experiments with the method nra^*_ϵ is very similar to a monotonic decreasing hyperbola within the range of 0% up to 30%.

On the one hand, nra^*_ϵ has better and more predictable run times than a^*_ϵ. However, a^*_ϵ achieves significant better qualities for ϵ in the range of 0.05 to 0.35 (but yields worse results for $\epsilon = 0.4$). Lemma 1 formulates the reason for the reduced quality of nra^*_ϵ: the path to an expanded state might be suboptimal and is never improved later since the state is not reopened again. I.e., for nra^*_ϵ, cost deviations on an optimal path are always *permanent*, while those for a^*_ϵ may be corrected later. However, the commentary to Theorem 5 formulated the expectation that the loss in quality would stay far below the stated bound. This is confirmed by the experimental results.

To explain the improved predictability of the new method nra^*_ϵ, also the number of state expansions and state reopenings has been acquired in the experiments. The percentage of expansions that reopen a state during a run of a^*_ϵ operating at different degrees of relaxation has been depicted in Fig. 9. As can be seen, the mean percentages of reopenings for the nine groups have a monotonic growth in ϵ. This explains the better run times of the nra^*_ϵ-method which is not decelerated by any reopening.

For both methods, the total run time increases again for a degree larger than 30%. Similar results have been observed in [70] where the *Traveling Salesman Problem* (TSP) has been used as a test vehicle for A^*_ϵ: there, as with our application, the number of states expanded often is not a monotonic decreasing function. The reason for this phenomenon lies with the modified condition for the selection of the next, most promising state. This directly influences the necessary condition for state expansion. While A* guarantees that no state q with $f(q) > C^*$ is expanded, a^*_ϵ (and with that, also nra^*_ϵ) only

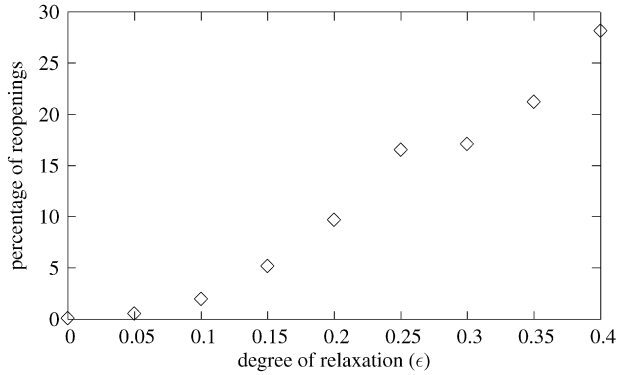


Fig. 9. Degree of relaxation vs. percentage of reopenings of a^*_ϵ.

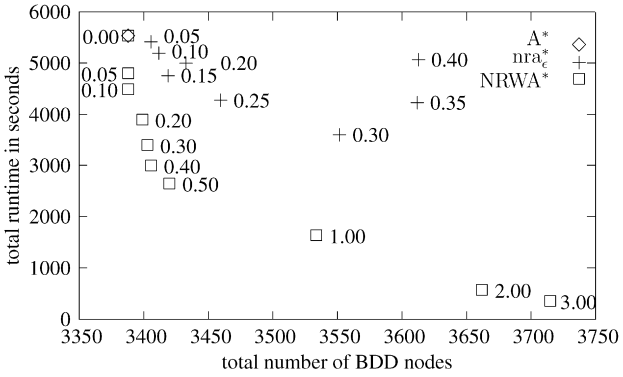


Fig. 10. Trading off run time for solution quality with nra^*_ϵ and NRWA*.

guarantees that states satisfying $f(q) > (1+\epsilon)\cdot C^*$ will be excluded from expansion. Consequently, it is possible that some states q satisfying the condition $(1+\epsilon)\cdot C^* \geqslant f(q) > C^*$ are expanded by weighted A*, but not by the original A* algorithm (also see Theorem 4). We have also experimentally verified that the number of state expansions first decreases as the degree of relaxation is raised, but later increases again. As the resulting plot was similar to the curve for nra^*_ϵ in Fig. 8, it has not been included due to space limitations.

The relaxation has a strong potential to reduce the run time. This is the positive effect of a more focused search. For a^*_ϵ however, at least two negative effects oppose this positive effect. The first is the increase of reopenings, the second is the potential for more state expansions in general. The unpredictability seems to result from the varying extent of influence of a particular relaxation ϵ on a total of three effects (one of them positive and two of them negative). In contrast, for nra^*_ϵ only one negative effect (the general potential for an increased number of state expansions) counteracts a reduction in run time caused by the relaxation. Here, the positive effect first predominates but eventually is absorbed and diminishes.

Consequently, the behavior is much more predictable than for a^*_ϵ. A disadvantage of nra^*_ϵ, however, are potentially worse results.

7.1.3. *Comparison of* nra^*_ϵ *to* NRWA*

In a second series of experiments A* and nra^*_ϵ have been compared to NRWA*. The results are depicted in Fig. 10. In contrast to the behavior of nra^*_ϵ, the run time of NRWA* is monotonic decreasing. This confirms the result of Theorem 4 as well as our expectations that also the revised version of WA*, i.e. NRWA*, would behave according to the upper bounds stated in Theorem 4 (as has been explained in the remarks to Corollary 2). For NRWA*, the degradation of solution quality first increases slowly (e.g., for $\epsilon \in [0, 0.5]$) and later ascends more steeply with increasing ϵ.

The next experiments confirm the observed improved suitability of NRWA* for a run time/quality trade-off by the user. When comparing the run time of NRWA* to that of A*, Fig. 11 illustrates how the gain in run time grows monotonic with the degree of relaxation (the curve in the space spanned by the percentage of gain and the degree ϵ is a convex hyperbola). At the higher relaxation degree of $\epsilon = 3.0$ the reduction in run time is already more than 90% on average.

Taking into account that NRWA* also has much more convenient theoretical properties than NRA^*_ϵ (in particular, NRWA* guarantees a much tighter upper bound for the deviation of the solution from the optimum), NRWA* seems to be superior to NRA^*_ϵ both from a theoretical and a practical standpoint. It should be noted however, that we have only limited practical evidence for this (in one domain only, BDD minimization). As Fig. 12 shows, high speed-ups can be obtained at an only

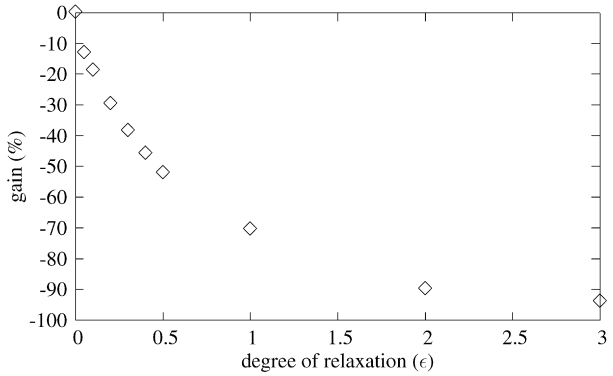


Fig. 11. Degree of relaxation vs. gain in run time of NRWA*.

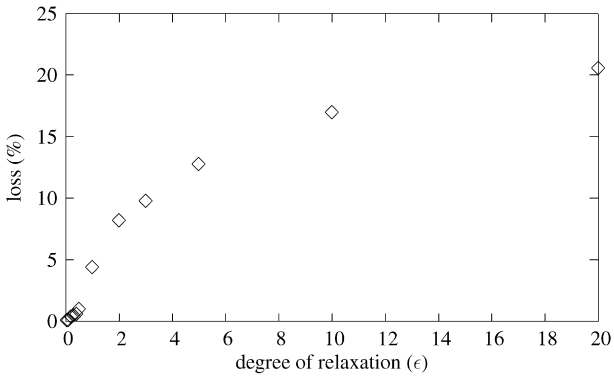


Fig. 12. Degree of relaxation vs. loss in quality of NRWA*.

small degradation of solution quality. In fact the average degradation is considerably less than the worst-case degradation as guaranteed by the theory (a factor of $1+\epsilon$). Also notice that for the first smaller weights the percentage of degradation grows slowly with the weight.

Similar results have been reported by Korf for the application of weighted A* to instances of the 15-puzzle [55]. This is also consistent with the results of Section 7.2, which describes experiments for typical planning domains. The results achieved by the method are much better than the quality guaranteed by the bound. On the one hand, this is a pleasant property for the practice. However, this also may indicate that the stated bound of $1+\epsilon$ can be tightened.

Operating at 40% of relaxation, on average the results are only 0.5% larger than the optimum BDD size. When theoretically allowing for solutions that are twice the minimum size, the average degradation still is only 4.3%. We also applied very high relaxations: Fig. 12 shows that the average degradation stays below 20% for a wide range of high relaxation degrees, it first reaches 20.5% for $\epsilon = 20.0$. For the higher weights, the resulting plot forms a convex hyperbola where the slope falls with ascending degree of relaxation.

7.1.4. *Comparison of* WA* *to* NRWA* *and* DWA* *to* NRDWA*

In a third series of experiments, the solution quality and run time of the algorithms WA* and NRWA* have been compared (see Fig. 13). In contrast to the strong improvement that has been observed for Algorithm nra^*_ϵ when compared to a^*_ϵ, the revised version of WA*, i.e. Algorithm NRWA*, shows no significant reductions in run time when compared to WA*. The reason is that, on average, reopenings are not a significant cause of run time for WA*. Different from the observations for nra^*_ϵ, the average number of reopenings does not grow significantly with the degree of relaxation.

However, the picture can change when looking at specific benchmarks: When using WA* for such examples as the circuit *s208.1*, the percentage of reopenings reaches 33.5% at a relaxation of $\epsilon = 2.0$. For *cm150a*, *mux*, even 52.7% of the state expansions have been due to reopenings at a relaxation of $\epsilon = 1.0$. Accordingly, the run times for these benchmark functions are significantly higher when using WA* instead of NRWA*. Therefore, NRWA* offers more stable run times than WA* since no reopenings can occur during a run of the algorithm. However, there may be a certain penalty in quality for the increased robustness: while there is no significant difference in the quality of solutions obtained by WA* and NRWA* for relaxation degrees up to 50%, at a relaxation of $\epsilon = 1.0$ the solutions obtained by NRWA* are on average 8.2% larger than the solutions yielded by WA*. This corresponds to the respective observation in Section 7.1.2: by Lemma 1, states might be expanded by NRWA* while the best known path to them is still suboptimal and, due to the modified behavior of NRWA*,

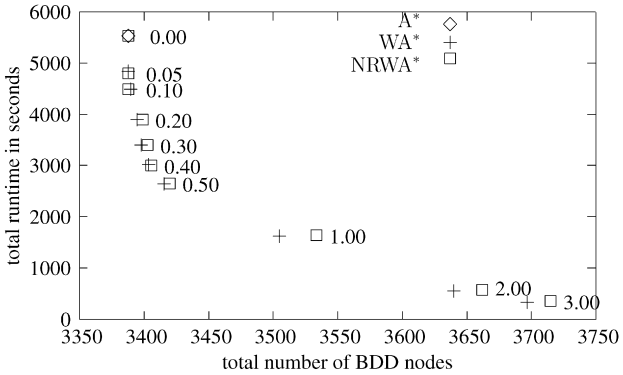


Fig. 13. Trading off run time for solution quality with WA* and NRWA*.

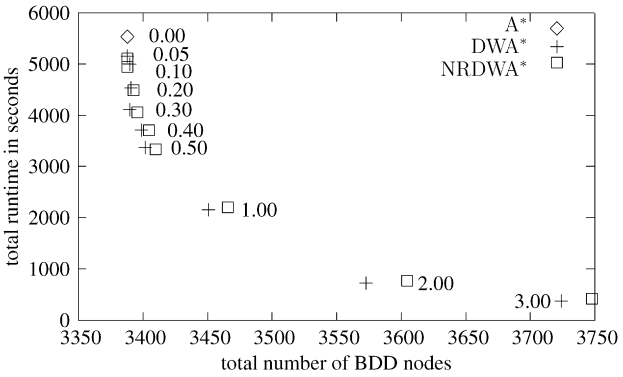


Fig. 14. Trading off run time for solution quality with DWA* and NRDWA*.

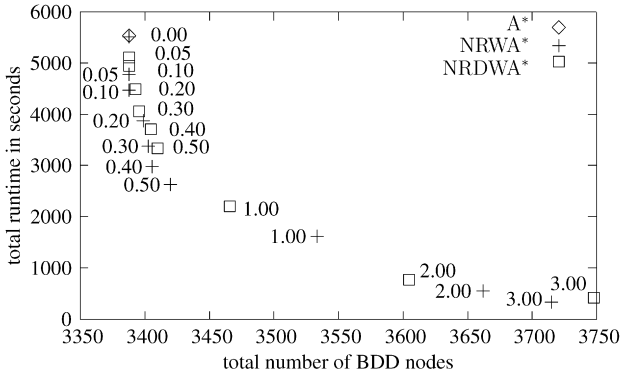


Fig. 15. Trading off run time for solution quality with NRWA* and NRDWA*.

no reopening/improvement can take place later. However, by a result of Likhachev et al., NRWA* yields a result whose cost does not exceed the optimum by more then a factor of $1 + \epsilon$ [63]. For this reason, the penalty for improved robustness remains comparatively small.

Second, almost the same observations have been made when comparing DWA* and NRDWA* (see Fig. 14).

7.1.5. *Comparison of* NRWA* *to* NRDWA*

Next, in a fourth series of experiments, NRWA* has been compared to NRDWA* in terms of quality and run time. As can be seen from Fig. 15, NRDWA* has significantly higher run times than NRWA* while at the same time slightly better results can be obtained. There is no clear relationship between the percentage of improvement of the solution and the degree of relaxation. However, at ascending degrees of relaxation, the blow-up of the average run time of NRDWA* increases. E.g., at a relaxation of $\epsilon = 0.3$, the run time of NRDWA* is 15.4% higher than that of NRWA* on average whereas the solution obtained by NRDWA* on average has 1.8% less BDD nodes than that of NRWA*. Further increase of the relaxation yields

Table 1
Results of NRWA*, evolutionary algorithm, simulated annealing and sifting.

$\sum$ time A*	$\sum$ opt	0.4		3.0		genetic		annealing		sifting	
		$\sum$ time	$\sum$ size	$\sum$ time	$\sum$ size	$\sum$ time	$\sum$ size	$\sum$ time	$\sum$ size	$\sum$ time	$\sum$ size
5508.39 s	3388	2978.5 s	3406	331.3 s	3715	8.05 s	3409	6.77 s	3411	$<$ 0.35 s	3701

improvements in the quality below 3% and yields ascending blow-ups of run time. At a relaxation of $\epsilon = 2.0$, the run time of NRDWA* is already 35.9% higher than that of NRWA* on average whereas the solution obtained by NRDWA* on average is 1.6% smaller than the solution of NRWA*. Summarized, there is a significant penalty for only small improvements in quality provided by NRDWA*.

7.1.6. Comparison of NRWA* *to classical simulation-based and heuristic techniques*

A fifth series of experiments compares the method NRWA* with classical simulation-based and heuristic techniques like *Evolutionary Algorithms* (EAs) [23], *Simulated Annealing* (SA) [4], and Rudell's sifting [76] which performs a classical hill-climbing approach. All algorithms are integrated into the CUDD package which is publicly available at [83]. The critical points of our results are shown in Table 1. Column $\sum$ *time* A* shows the total run time for computation of the minimum BDD size for 28 benchmark circuits of the LGSynth93 test suite with A*. Column $\sum$ *opt* shows the total number of BDD nodes of all BDDs needed for the minimal representation of the 28 circuits. The following five double-columns (0.4), (3.0), *genetic*, *annealing*, and *sifting* show the total run time ($\sum$ *time*) and the total number of BDD nodes ($\sum$ *size*) of the BDDs resulting from

- the method NRWA* running at a degree of relaxation of $\epsilon = 0.4$ and $\epsilon = 3.0$, respectively,
- the genetic algorithm of [23],
- the simulated annealing approach of [4],
- the hill-climbing approach of [76].

Using NRWA* at a relaxation degree of 40%, a large reduction of run time of 45.9% can be observed when comparing with A*. The actual average degradation in quality at this degree of relaxation is only 0.53%, which is a better average quality than with the EA and with SA. For the more complex circuits, the solutions obtained by the EA, SA, and sifting can show a significant degradation. E.g., for *comp*, the solution shows almost 10% more BDD nodes than the optimum size when using SA, more than 20% when using the EA, and sifting results in an almost 50% blow up of the solution.

In contrast, the method NRWA* has the advantage of a guaranteed upper bound for the deviation from the optimum. This does not come for free: the run time is very high compared to the simulation-based or heuristic methods. However, with the theoretical nonapproximability result of [81], this of course is an expected result.

If an average quality as low as for sifting is acceptable, much smaller run times can be achieved by higher degrees of relaxation. At a degree of $\epsilon = 3.0$, all benchmarks of the test suite can be minimized within a few minutes. A certain advantage over sifting is that the cost of the results is still guaranteed to be within four times the optimum, an upper bound for the deviation which sifting as well as the EA or SA do not guarantee. It can be expected that this bound may even be grossly exceeded by EA/SA for hard problem instances, i.e. the benchmarks that cannot be solved by A* (albeit first experiments showed no evidence which supports this hypothesis, see Section 7.3). Within the limited scope of our experiments, the simulation-based approaches performed well enough to raise the question whether the use of weighted A* is always beneficial in the domain of BDD minimization. On the other hand, EAs/SA might not always work that well for the hard examples of different benchmark suites.

Moreover, good simulation-based approaches mostly depend on domain knowledge (this holds in particular for the fitness functions, cross over and mutation operators of EAs) and they might not be at hand for every application domain. In particular when solutions are very sparse, many local search methods run into problems. Opposed to that, there exists a *domain-independent* realization of A* and weighted A* (see Section 7.2).

7.2. Weighted A* *and* STRIPS *planning*

This section describes the application of weighted A* to STRIPS benchmark problems of typical planning domains such as Blocksworld, Puzzle (the sliding-tile puzzle also known as $(n \times n) - 1$ puzzle), Depots, Logistics, PSR, Satellite, Freecell, and the Driverlog domain. The additional experiments put the observations in Section 7.1 into a wider context of different domains, each of which with its own characteristics.

7.2.1. Background

In the past, several domain-independent heuristic problem solvers have been suggested, including the optimal planners STAN [65], BLACKBOX [50], and HSP (and, more recently, HSPr/HSP2) by Bonet and Geffner [6,7]. The more recent versions HSPr and HSP2 solve STRIPS planning problems by use of weighted A*, and they are both able to search backward from the goal to the initial state (known as "regression search" [69,85]). This was a significant improvement over the first version of

HSP since that way unnecessary recomputations of the heuristic are avoided [6]. The heuristic is the domain-independent and admissible heuristic called "max-pair" (denoted h^2) by Haslum and Geffner [41].

HSPr showed a good performance in the biennial ICAPS planning competitions [6]. The source code has been made available to the public. The latest version called HSP2 subsumes the functionalities of the previous releases HSP and HSPr. For our experiments, an implementation of NRWA* has been obtained by appropriate modifications of HSP2. Throughout the experiments, backward search and the "max-pair" heuristic was used.

The application of A* and WA* to STRIPS problems has been subject to previous work by Bonet and Geffner [7] and by Hansen and Zhou [38]. The present experiments extend these results in two aspects: first, the variation of the weight has not been studied very intensively. In [7] only three weights (1.0, 2.0, and 5.0) have been applied. Throughout the experiments in [38], one more weight, 10.0, has been applied. The same authors also examine the idea of decreasing the weight after each iteration of their Anytime WA* (AWA*) algorithm. The reported total run times are given for the completion of all iterations. While shedding light on the role of weights within an anytime framework, it is difficult to derive which weight would be good for the original WA* approach in a particular domain. Second, NRWA* so far has only been analyzed within the framework of Anytime algorithms: Likhachev et al. applied their ARA* algorithm with a special technique to limit node reexpansions to one particular domain (a robot planning domain) [62,63]. Hansen and Zhou reimplemented ARA* and examined Likhachev's idea to limit the number of reexpansions, with regard to more than one domain [38].

Different from the first suite of experiments, we did not include the remaining variants of WA* as both DWA* and A^*_ϵ depend on the application domain. Dynamic Weighting works with an upper bound on an optimal solution length N. For some domains it is not hard to give this number (e.g., BDD minimization or Pohl's original problem TSP), for others it may be much harder. The automated derivation of a bound for arbitrary STRIPS domains seems very difficult. Even when given one upper bound, the use of a tighter bound already suffices to change the behavior (see [72]).

As has been discussed in Section 3, Pearl and Kims' method A^*_ϵ leaves a high degree of freedom. It is advantageous to utilize this freedom using the available domain knowledge. On the one hand, it is possible to obtain a domain-independent variant of A^*_ϵ, e.g. by using the "max-pair" heuristic h^2 and by resorting to the "focal" heuristic $h_F = h^2$. However, this way the full potential of A^*_ϵ is hardly used and this hinders a fair comparison to truly domain-independent methods.

Due to the space restrictions, we also did not include local search methods. Instead we would like to refer to the respective previous work, e.g. the experiments of Bonet and Geffner where a domain-independent hill-climbing approach [6,7] was applied to STRIPS problems in the same and similar planning domains.

7.2.2. Results

The results of the conducted experiments are depicted in Figs. 16(a)–20(b). Figs. 16(a)–20(b), and Figs. B.3(a), B.3(b) in Appendix B, respectively, show the progression of the average run time and loss in quality with the degree of relaxation ϵ for the benchmark problems in the respective domain. Figs. B.1, B.2(a), and B.2(b) in Appendix B show how the total number of expanded nodes for the problems in the respective domain varies with ϵ.

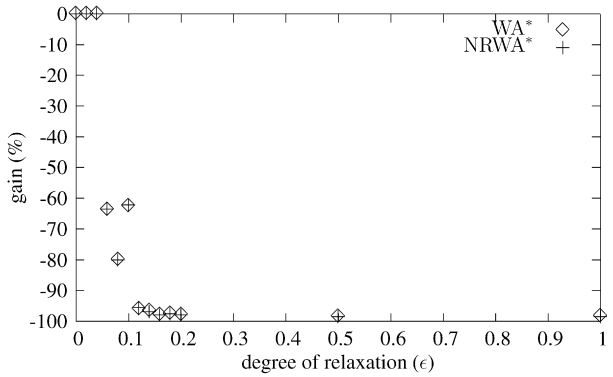
As the results show, weighted A* generally works very well for the majority of the examined domains. For some domains, a large gain in run time is already observed for weights smaller than 1.20: e.g. for the Blocksworld domain, a weight as small as 1.08 already yields a reduction in run time of 80%, quickly reaching 98.0% for a weight of 1.20 (see Fig. 16(a)).

Other domains require slightly larger weights to achieve the best possible run time reductions: for the Logistics domain, the observed gain for a weight of 1.20 is already more than 40% percent. But the gain can be increased to more than 90% when a weight of 1.50 is applied (see Fig. 17). In the Satellite domain, the maximum reduction in run time was 60% (achieved for a weight of 2.0 at an average loss in quality of 6%, see Figs. 18(a) and 18(b)).

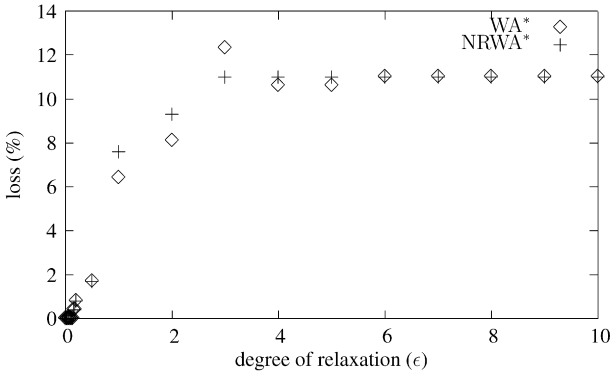
The gain curves generally show a high steepness, except for the PSR domain, where the curve is flatter. Here, a higher reduction in run time of more than 80% is first observed for a weight of 3.0 (but the reduction increases to more than 90% for weights beyond 3.0, see Fig. 19). The curves for the gain in run time are monotonically decreasing and relatively smooth for almost the total range of ϵ in all domains. The presented theory gives reasons for this overall convenient behavior (see Theorem 4 in Section 3). On the other hand there is no theoretical reason why this should always be the case, and actually, two noteworthy exceptions have been encountered (see Experiment 1 in Appendix B).

The Blocksworld domain shows a relatively smooth degradation of quality with increasing weight (see Fig. 16(b)). For other domains, the curves show several plateaux that are reached for the different ranges of weights. For two domains, there is only one such plateau at zero %, i.e. for all considered weights the quality is fully preserved. This holds even when applying high weights (Logistics and PSR, see Figs. 17 and 19). For all domains the loss in quality is either zero % or is very close to zero if an appropriate weight is chosen. At the same time, such a weight yields a very high reduction in run time, i.e., at least a 10X speedup. With one exception, the use of high weights did not yield losses more than 6% to 12% on average, and also the average run time did not increase (for an exception see Experiment 2 in Appendix B). As far as the performance resulting from the use of high weights is concerned, the actual run times are determined by the interaction of a positive effect (a more focused search) and a negative effect (more reopenings, see Experiment 3 in Appendix B).

In all experiments, weighted A* seems to reach a "fixed point" where further increases of the weight neither have any significant impact on the behavior nor on the results of the run of the algorithm. Instead it seems that the same portion of the search space is visited and the same solution is found regardless of further weight increases. This is consistent with the respective observation in the BDD minimization domain (see Figs. 11 and 12). There is a simple reason: with large weights



(a) Gain for Blocksworld domain.



(b) Loss for Blocksworld domain.

Fig. 16. Gain and loss for the Blocksworld domain.

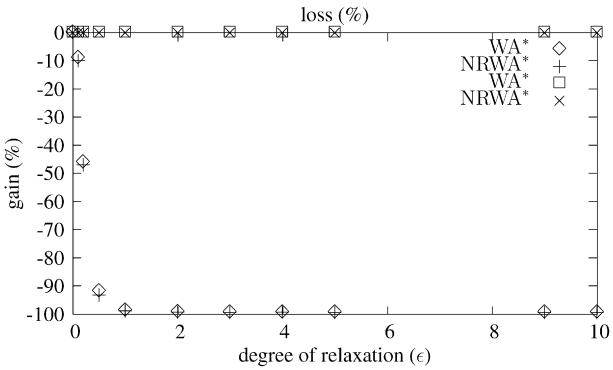
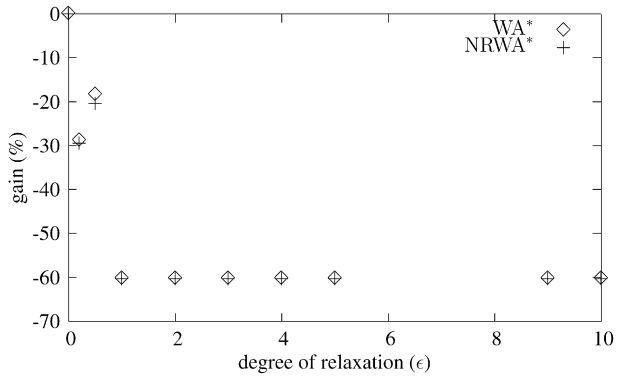


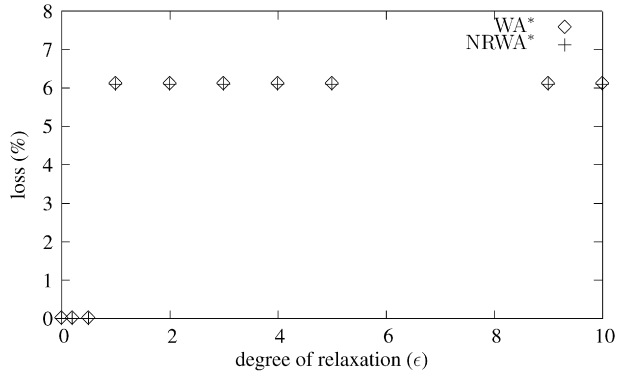
Fig. 17. Gain and loss for the Logistics domain.

$1+\epsilon$, the term $(1+\epsilon)\cdot h$ dominates the g-value, which at some point has no effect on the decisions of the algorithm at all. By this, weighted A* gradually fades to greedy best-first search (see Section 2.1).

There are also two examples where WA* did not work very well (see Experiment 4 in Appendix B). When comparing NRWA* to WA*, only very small average gains are observed. However, when looking at specific problems, the method can yield significant gains in run time: Experiment 4 in Appendix B, conducted in the Freecell domain (with a reduction in the number of expanded and generated nodes of almost 15%, without any loss in quality) is one of them. Using a weight of 10.0 in the Puzzle domain, NRWA* expands 30% less nodes and generates 28% less nodes than WA* for problem "prob03". Again, there is no penalty in terms of quality loss. For other problems, the observed reduction in the number of generated and expanded nodes was smaller (around 1–10%). However, there is also (only) one example where NRWA* yields a higher run time than original weighted A* (see Experiment 5 in Appendix B). For more appropriate weights, this negative effect again vanishes completely.



(a) Gain for Satellite domain.



(b) Loss for Satellite domain.

Fig. 18. Gain and loss for the Satellite domain.

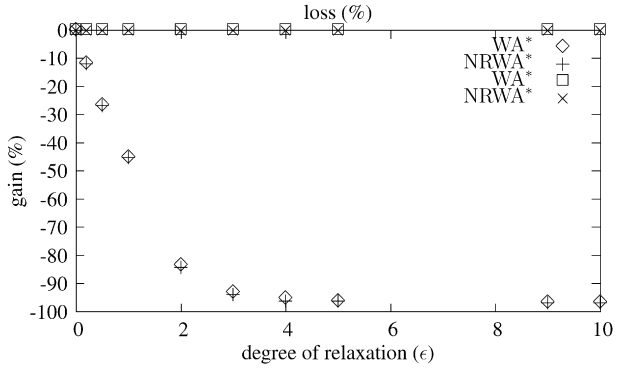
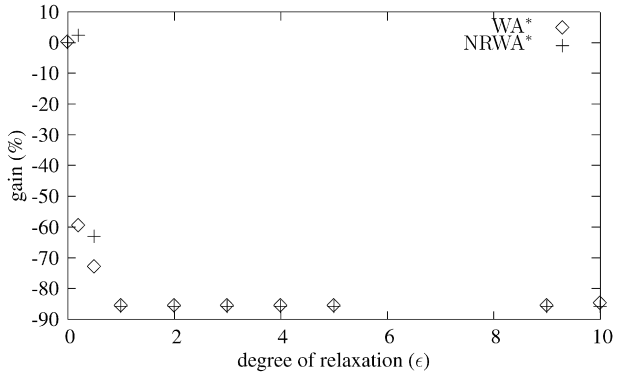


Fig. 19. Gain and loss for the PSR domain.

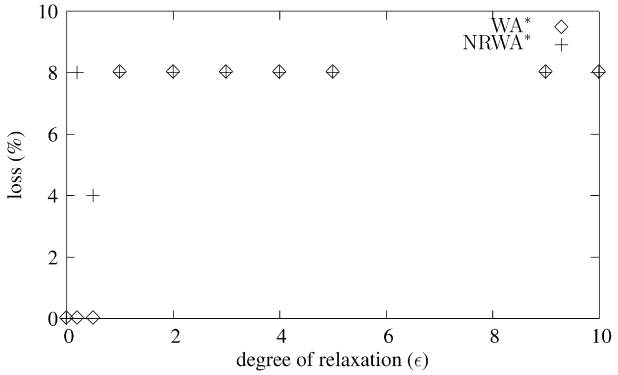
NRWA* sometimes has a slightly higher loss in quality than WA*. This seems to happen only when an inappropriate weight has been chosen (examples are the Blocksworld, the Depots and the Puzzle domain, see Figs. 16(b), 20(b), and B.3(b)). Otherwise, the quality is as good as with WA*.

7.3. Hard problem instances

The experiments with weighted A* described in the previous sections included a comparison to the results and the performance of the original A* algorithm. Therefore it was necessary to restrict the test suites to those cases that could be solved with A*. Besides that, it is also interesting to see whether hard instances (i.e., problems that cannot be solved with A* within a given time and memory limit) can be solved with weighted A*. In a last series of experiments, we applied a weight of 1.20 to the hard instances of the considered STRIPS domains. On the one hand, applying weights larger than 1.20 would possibly yield more solutions. However, the obtained solution lengths would not be guaranteed to be at most 20% away from the optimum anymore, making it more difficult to interpret the results. In two domains, Blocksworld and PSR,



(a) Gain for Depots domain.



(b) Loss for Depots domain.

Fig. 20. Gain and loss for the Depots domain.

Table 2
Results of WA* ($\epsilon = 0.2$) applied to hard instances.

problem	solution length	# generated nodes	# expanded nodes
BLOCK-14-0	38	101281	33416
PSR-P50_S107	23	31244	20451

solutions for hard instances have been found. In Table 2, the results for the largest solved hard instances are given (solution lengths and node counts were identical for WA* and NRWA*).

For the BDD domain, the solutions for some hard instances have been compared to the results of sifting and the simulation-based approaches. For a weight of 1.20, several hard instances could be solved, among the largest are *count* and *term1* with 35 and 34 inputs, respectively. The results were not better than those for sifting. For a weight of 2.0, a solution for *term1* of 90 BDD nodes has been obtained within a few minutes. The result of sifting is 163 BDD nodes, which is more than 80% higher. However, the results of the simulated annealing approach and the genetic algorithm are better: SA results in 85 BDD nodes and the EA yields only 75 BDD nodes, both within a few seconds.

7.4. Summary of experimental results

For the domain of BDD minimization, the methods a^*_ϵ, nra^*_ϵ and NRDWA* have been included in a comparison to WA* and its non-reopening variant NRWA*, while we focused on (domain-independent) realizations of the latter two approaches for the STRIPS problems in several planning domains.

In contrast to the rather erratic behavior of a^*_ϵ for our application, the new method nra^*_ϵ shows a much more predictable behavior. A practitioner can use nra^*_ϵ to achieve the desired run time/quality trade-off at relaxation degrees up to 30%. When compared to our instantiation of Pearl and Kims' A^*_ϵ, the increased predictability and the better run times come at the cost of a reduction in quality. If solution quality is what matters most, the original method of Pearl and Kim can be a better choice than the non-reopening variant.

On the one hand, the method NRWA* can be considered superior to NRA^*_ϵ both from a theoretical and practical standpoint: first, the practitioner can rely on the much tighter bound of NRWA*. For this reason also the qualities of NRWA* were much better than those of nra^*_ϵ. Second, NRWA* did not show a turning point in run time, i.e. no relaxation degree has

been observed where the number of state expansions (and therefore, the run time) would have increased again. Theorem 4 sheds light on the reason for the better behavior.

However, the degrees of freedom in the approach of Kim and Pearl allow for a better utilization of an expert's domain knowledge. The fact that using this freedom was not beneficial in our BDD minimization domain (implemented with one particular "focal" heuristic) does not imply that this will also be the case for other domains. Also notice that, by the results of Section 3, NRWA* is just an instance of the general framework provided by A^*_ϵ.

The advantages of NRDWA* over NRWA* are debatable: slightly better results come at a rather high penalty in run time. Concerning average run time, our experiments in the BDD minimization domain did not show a clear advantage of using NRWA* instead of WA* because the method WA* did not suffer as much from reopenings as a^*_ϵ did. Similar results have been obtained for the planning domains. For NRWA*, the reduction in the number of node generations and expansions is as low as one percent on average, but can be up to 30% for specific problems. In the BDD domain, the corresponding observed maximum gain of NRWA* in comparison to WA* was higher, more than 50%.

There was only one example where the use of NRWA* caused a high increase in the number of node generations and expansions (the Depots domain, using a weight of 1.20 or 1.50) and therefore the run time here was worse than that of WA*. In the BDD minimization as well as in the planning domains for some weights a small penalty in terms of quality can been observed when using NRWA*. Nevertheless, given, that a more appropriate weight is used, there is no example of a significant penalty in run time or quality for not reopening expanded nodes.

In the BDD minimization domain, NRWA* has also been compared with local search techniques. The practitioner can choose an expected quality similar to that of hill-climbing approaches like sifting. In this case NRWA* processed the whole test suite in minutes (however, the guaranteed bound for the deviation from the optimum then is already a factor of four). If a (guaranteed) higher quality is needed for the targeted application, the run times of weighted A* are still at least an order of magnitude away from that of sifting and the simulation-based methods. Hence, if and only if unpredictable outliers in the results or even frequently downgraded qualities (as, in particular, yielded by sifting) can always be tolerated by the application, sifting would probably be the way to go. If, in addition, higher run times can be accepted, the use of EAs reduces the expected frequency and magnitude of deviations from the optimum significantly. However, this still has the disadvantage of an unpredictable quality of the results. Moreover, the design of a good EA depends on domain knowledge and might not be at hand for every application domain, particularly if solutions are sparse.

For the remaining applications where a bounded suboptimality is required (as the aforementioned VLSI applications, see Section 6), even very high run times can be accepted. Here, weighted A* search seems to offer an interesting alternative, which, as a bonus, can be implemented as a domain-independent approach.

Summarized, weighted A* works very well for most of the considered problems, allowing for reductions in run time of up to 98%, at a zero or a very small loss in quality. Exceptions are most of the problems in the Freecell domain and one problem in the Driverlog domain. With regard to the different variants of weighted A*, there is no definite winner. The same seems to hold when comparing weighted A* with local search approaches. Depending on the different users' preferences, one method might be better suited than another. Theorem 4 formulates an advantage of constant overweighting over an A^*_ϵ algorithm that uses a standard cost function $f = g + h$. There also is some empirical evidence for this (obtained from experiments in one particular domain, BDD minimization). Nevertheless, the advantages of the A^*_ϵ framework, namely its freedom and generality, remain (see Section 3).

Some domains require higher weights than others. Among the considered domains, the domain with the smallest effective weights is the Blocksworld domain, the one with the highest is the PSR domain. PSR as well as the BDD minimization domain require higher weights than the remaining planning domains. While it seems difficult to give a general advice for the choice of appropriate weights, our results suggest to choose a comparatively high weight as the first trial. It is worth mentioning that the choice of the default weight of 2.0 in the HSP planner of Bonet and Geffner also follows this strategy. In our experiments, this suffices to achieve a high reduction in run time in most cases. However, when applying even higher weights, a loss in quality of up to 20% (for one example, the Puzzle domain, even up to 70%) is the penalty. Therefore it can be beneficial to repeatedly apply gradually decreased weights until the sum of the solution lengths falls below a threshold for quality (or, alternatively, a threshold for the relative improvement of quality).

8. Conclusion

We have presented a unifying view on previous approaches to weighted A*. A particular concern of the paper was the effect of not reopening expanded states. This also led to some novel variations. All considered approaches have been studied from a theoretical and an empirical perspective. As a technical contribution, a novel general bound on suboptimality for weighted A* has been derived from the unifying view. In an experimental evaluation, the BDD minimization problems corresponding to benchmark circuits and STRIPS benchmark problems of several classical AI planning domains have been solved by the respective variants of weighted A*. The experiments clearly demonstrate the efficiency of weighted A*.

On the one hand, the actual performance depends on the problem domain and the problem instance. In this, different impacts of weight and reopenings can be observed. On the other hand, many aspects of the behavior were found to be similar in all considered domains.

With regard to the considered variants of weighted A*, there was no definite winner as they differed in performance, presenting more than one tradeoff. Besides the essential tradeoff run time vs. quality other tradeoffs were considered,

e.g. predictability of behavior vs. quality. Weighted A* was also compared with several local search approaches. Again, it turned out that which method suits best depends on the preferences of the user.

We provided a detailed discussion of our experiences with weighted A*, which is of value for the AI practitioner.

Acknowledgements

The authors would like to thank the anonymous reviewers for their helpful comments on an earlier draft of this paper. We would like to thank Shavila Devi, Armando Franco, Brandon Jue, and Jennifer Worley for their help with the English presentation. Our thanks also go to Laura Lo and Yun-Pang Wang for the coordination of the proofreading. Finally, we thank Tony Cohn who, after the coordination of the reviewing process, also found the time to make valuable suggestions to further improve the English presentation of the final version.

Appendix A. Proofs

Proof of Theorem 2. Let $q_0 = \arg\min_{q \in \text{OPEN}} f(q)$. We have

$$f(\hat{q}) \leqslant f'(\hat{q}) \tag{A.1}$$
$$\leqslant f'(q_0) \tag{A.2}$$
$$\leqslant f^{\Uparrow}(q_0) \tag{A.3}$$
$$= (1+\epsilon) \cdot f(q_0)$$
$$= (1+\epsilon) \cdot \min_{q \in \text{OPEN}} f(q) \tag{A.4}$$

Eq. (A.1) holds by the definition of f' in the assumption. Next, Eq. (A.2) holds by definition of $\hat{q}$. Then, Eq. (A.3) holds again with the definition of f'. By Eq. (A.4), $\hat{q} \in$ FOCAL already follows. □

Proof of Theorem 3. First it is easily verified that

$$f(q) \leqslant f^{\text{DW}}(q) \leqslant f^{\uparrow}(q) \leqslant f^{\Uparrow}(q)$$

for all states q of the considered state space. By Theorem 2, the respective next state expanded by WA* and DWA* must be contained in FOCAL. Second, A^*_ϵ chooses a state q_F from FOCAL with $q_F = \arg\min_{q \in \text{FOCAL}} h_F(q)$. As h_F is assigned to the respective cost function, and since the same respective tie-breaking rule is used, A^*_ϵ must act exactly as DWA* and WA*, respectively. □

Proof of Theorem 4. The results for the cases $\text{A} = \text{A}^*, \text{A}^*_\epsilon$ are already well-known [40,70]. They are included to compare them to the new results. Because q is expanded before q',

$$f^{\uparrow}(q) = f(q) + \epsilon \cdot h(q) \leqslant f^{\uparrow}(q') \tag{A.5}$$

in the case $\text{A} = \text{WA}^*$, and

$$f^{\text{DW}}(q) = f(q) + \epsilon \cdot \left[1 - \frac{d(q)}{N}\right] \cdot h(q) \leqslant f^{\text{DW}}(q') \tag{A.6}$$

in the case of $\text{A} = \text{DWA}^*$. To derive the stated upper bounds for $\text{A} = \text{WA}^*$, $\text{A} = \text{DWA}^*$, it now suffices to separate $f(q)$ on the left side of the two equations (A.5) and (A.6), respectively. The upper bounds range within the stated intervals since

- the term $h(q')$ can be bounded by $h^*(q')$ because of the admissibility of h, and
- since an optimal path is considered, we have $g(q') = g^*(q')$ and finally $f^*(q') \leqslant C^*$. □

Proof of Lemma 1. Consider an optimal path p from s to q. Let q' be the first state on $p = s, \ldots, q', \ldots, q$ which also appears on OPEN.[4] Assume that $q \neq q'$ and that q is selected for expansion. In A* this implies $f(q) \leqslant f(q')$ and, using the monotonicity of h (see Eq. (2)), it is straightforward to conclude the result of Theorem 1 $g(q) = g^*(q)$. However, in A^*_ϵ the situation is different due to the relaxed selection condition.

Let f_0 be the minimal cost of a state on OPEN. With Eq. (3) and the selection condition it is $f(q) \leqslant (1+\epsilon) \cdot f_0$. On the other hand we have $f_0 \leqslant f(q')$ by definition of f_0. Thus, $f(q) \leqslant (1+\epsilon) \cdot f(q')$. Different from the situation in A*, the cost of q can exceed that of q' (by a factor of $1+\epsilon$ in the worst-case).

[4] Notice that it is straightforward to prove that, during operation of the algorithm, at least one state on p must be an open state. The proof is an induction on the length of p which is started by s, the very first state occurring both on OPEN and p.

Consequently,

$$\begin{aligned} g(q) + h(q) &\leqslant (1+\epsilon) \cdot \big(g(q') + h(q')\big) \\ &= (1+\epsilon) \cdot \big(g^*(q') + h(q')\big) && \text{(A.7)} \\ &\leqslant (1+\epsilon) \cdot \big(g^*(q') + k(q', q) + h(q)\big) && \text{(A.8)} \\ &= (1+\epsilon) \cdot \big(g^*(q) + h(q)\big) && \text{(A.9)} \end{aligned}$$

Eq. (A.7) holds: since q' is the first open state on an optimal path p where all ancestors of q' along p are closed, $g(q') = g^*(q')$. Eq. (A.8) holds with the monotonicity of h, see Eq. (2). Eq. (A.9) holds since the sum $g^*(q') + k(q', q)$ is equal to $g^*(q)$ because q' is an ancestor of q along an optimal path p. Then Eq. (4) follows after separating $g(q)$ and $g^*(q)$ on the left side of the equation.

Next, let us consider the operation of WA*. Since q is expanded before q', we have $f^{\uparrow}(q) \leqslant f^{\uparrow}(q')$, and consequently, with a similar line of argument as before,

$$\begin{aligned} g(q) + (1+\epsilon) \cdot h(q) &\leqslant g(q') + (1+\epsilon) \cdot h(q') \\ &\leqslant g(q') + (1+\epsilon) \cdot \big(k(q', q) + h(q)\big) \\ &= g^*(q') + k(q', q) + \epsilon \cdot k(q', q) + (1+\epsilon) \cdot h(q) \\ &= g^*(q) + \epsilon \cdot k(q', q) + (1+\epsilon) \cdot h(q) \end{aligned}$$

Consequently, $g(q) \leqslant g^*(q) + \epsilon \cdot k(q', q)$ and Eq. (5) follows. □

Proof of Theorem 5. Consider an optimal path p from s to q_1. Let q'_1 be the first state on $p = s, \ldots, q'_1, \ldots, q_1$ which also appears on Open. Assume that $q_1 \neq q'_1$ and that q_1 is selected for expansion. Due to the modified behavior of NRA^*_ϵ, q_1 is marked as closed afterwards. To start an induction on the length of p, assume that this is the first time this situation occurs during operation. Let f_0 be the minimal cost of a state on Open. As long as q'_1 resides on Open, we have

$$f(r) \leqslant (1+\epsilon) \cdot f_0 \leqslant (1+\epsilon) \cdot f(q'_1)$$

for every open state r that is eligible to expansion. Moreover,

$$f(q'_1) \leqslant C^* \qquad \text{(A.10)}$$

can be concluded: since q'_1 is the first open state on an optimal path, it is $g(q'_1) = g^*(q'_1)$. As h is admissible, Eq. (A.10) follows. Consequently, $f(r) \leqslant (1+\epsilon) \cdot C^*$.

However, the deviation can become worse: q'_1 might eventually be expanded as one of the next steps of operation. As NRA^*_ϵ ignores better paths to closed states, $g(q_1)$ will *not* be updated with the optimal path cost along q'_1 on p, i.e. $f(q_1)$ will never reach $f^*(q_1)$. However, by Lemma 1 this loss in exactness must be bounded,[5] i.e. it is

$$\begin{aligned} g(q_1) - g^*(q_1) &\leqslant \epsilon \cdot \big(g^*(q_1) + h(q_1)\big) \\ &\leqslant \epsilon \cdot C^* && \text{(A.11)} \end{aligned}$$

The last equation holds with a similar argument as for $f(q'_1)$ above.

In the following, refer to Fig. 1(b) for the notation and an illustration of the idea. In an induction on the length of p, a similar argument is applied repeatedly on all remaining pairs of adjacent states $({q_i}', q_i)$ on p, that is, for $1 \leqslant i \leqslant \lfloor \frac{N}{2} \rfloor$ we claim

$$g(q_i) - g^*(q_i) \leqslant \epsilon \cdot C^* \cdot \sum_{k=0}^{i-1} (1+\epsilon)^k \qquad \text{(A.12)}$$

In the case of $i = 1$, the claim is equivalent to Eq. (A.11). Now assume the claim is proven for i. For the step $i \longrightarrow i+1$ we derive

$$\begin{aligned} g(q'_{i+1}) &= g(q_i) + k(q_i, q'_{i+1}) \\ &\leqslant g^*(q_i) + \epsilon \cdot C^* \cdot \sum_{k=0}^{i-1} (1+\epsilon)^k + k(q_i, q'_{i+1}) && \text{(A.13)} \\ &\leqslant g^*(q'_{i+1}) + \epsilon \cdot C^* \cdot \sum_{k=0}^{i-1} (1+\epsilon)^k && \text{(A.14)} \end{aligned}$$

[5] The lemma can be applied here since the operation of A^*_ϵ and NRA^*_ϵ is identical before the first state has been reopened.

Eq. (A.13) holds with the induction hypothesis in Eq. (A.12), Eq. (A.14) holds with the optimality of p. It is $f(q_{i+1}) \leqslant (1+\epsilon)\cdot f(q'_{i+1})$ since q_{i+1} is expanded before q'_{i+1}: this holds with the same argument as applied before to q'_1. But then, similar to the argument of Lemma 1,

$$\begin{aligned}
\big(g(q_{i+1})+h(q_{i+1})\big) &\leqslant (1+\epsilon)\cdot\big(g(q'_{i+1})+h(q'_{i+1})\big)\\
&\leqslant (1+\epsilon)\cdot\left[g^*(q'_{i+1})+\epsilon\cdot C^*\cdot\sum_{k=0}^{i-1}(1+\epsilon)^k+h(q'_{i+1})\right]\\
&\leqslant (1+\epsilon)\cdot\left[g^*(q'_{i+1})+k(q'_{i+1},q_{i+1})+h(q_{i+1})+\epsilon\cdot C^*\cdot\sum_{k=0}^{i-1}(1+\epsilon)^k\right]\\
&\leqslant (1+\epsilon)\cdot\left[g^*(q_{i+1})+h(q_{i+1})+\epsilon\cdot C^*\cdot\sum_{k=0}^{i-1}(1+\epsilon)^k\right]
\end{aligned}$$

Eq. (A.15) holds with Eq. (A.14), Eq. (A.15) holds with the monotonicity of h (see Eq. (2)), and Eq. (A.15) holds with the optimality of path p. By Eq. (A.15), it is

$$g(q_{i+1})\leqslant(1+\epsilon)\cdot\left[g^*(q_{i+1})+\epsilon\cdot C^*\cdot\sum_{k=0}^{i-1}(1+\epsilon)^k\right]+\epsilon\cdot h(q_{i+1})$$

and consequently

$$\begin{aligned}
g(q_{i+1})-g^*(q_{i+1}) &\leqslant \epsilon\cdot\big[g^*(q_{i+1})+h(q_{i+1})\big]+(1+\epsilon)\cdot\epsilon\cdot C^*\cdot\sum_{k=0}^{i-1}(1+\epsilon)^k\\
&=\epsilon\cdot\big[g^*(q_{i+1})+h(q_{i+1})\big]+\epsilon\cdot C^*\cdot\sum_{k=0}^{i-1}(1+\epsilon)^{k+1}\\
&=\epsilon\cdot\big[g^*(q_{i+1})+h(q_{i+1})\big]+\epsilon\cdot C^*\cdot\sum_{k=1}^{i}(1+\epsilon)^k\\
&\leqslant\epsilon\cdot C^*+\epsilon\cdot C^*\cdot\sum_{k=1}^{i}(1+\epsilon)^k\\
&=\epsilon\cdot C^*\cdot\sum_{k=0}^{i}(1+\epsilon)^k
\end{aligned}\tag{A.15}$$

and the claim of the induction is shown. Eq. (A.15) holds with the admissibility of h. In particular, for $q_{\text{last}}=q_{\lfloor\frac{N}{2}\rfloor}$

$$g(q_{\text{last}})-g^*(q_{\text{last}})\leqslant\epsilon\cdot C^*\sum_{k=0}^{\lfloor\frac{N}{2}\rfloor-1}(1+\epsilon)^k$$

which is 0 if $\epsilon=0$, otherwise we have

$$\begin{aligned}
g(q_{\text{last}})-g^*(q_{\text{last}}) &\leqslant \epsilon\cdot C^*\sum_{k=0}^{\lfloor\frac{N}{2}\rfloor-1}(1+\epsilon)^k\\
&=\epsilon\cdot C^*\cdot\frac{(1+\epsilon)^{\lfloor\frac{N}{2}\rfloor}-1}{\epsilon}\\
&=C^*\cdot\big[(1+\epsilon)^{\lfloor\frac{N}{2}\rfloor}-1\big]\\
&=C^*\cdot(1+\epsilon)^{\lfloor\frac{N}{2}\rfloor}-C^*
\end{aligned}\tag{A.16}$$

Eq. (A.16) holds with the well-known sum formula for geometric series (which only applies in the case $\epsilon\neq 0$). That is, on any optimal path constructed during operation of NRA^*_ϵ, the deviation from the optimum is not greater than a factor

$$\frac{C^*+g(q_{\text{last}})-g^*(q_{\text{last}})}{C^*}=(1+\epsilon)^{\lfloor\frac{N}{2}\rfloor}\qquad\square$$

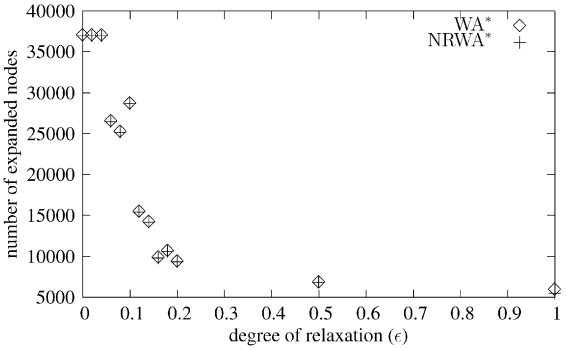


Fig. B.1. Total number of expansions for the Blocksworld domain.

Appendix B. Additional experiments

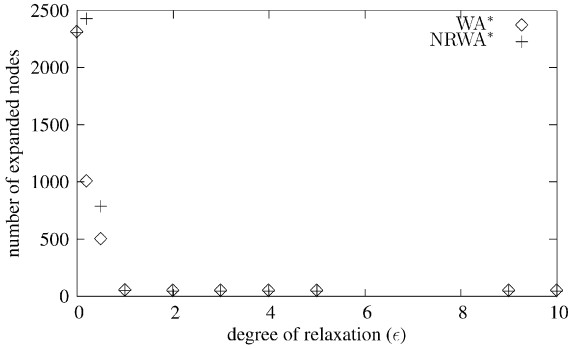
Experiment 1. For the Blocksworld domain there is one weight, 1.10, for which the curve progression suggests an average gain of around 90%, but the measured value is only 60% (see Fig. 16(a)). This corresponds to high numbers of expanded nodes that do not fit the general progression of the curve (see Fig. B.1). A similar observation has been made for the Satellite domain from IPC4 (see Fig. 18(a)). Here, the weight 1.50 yields a higher average run time and a larger total number of generated and expanded nodes than a weight of 1.20 (see Fig. B.2(b) for the expanded nodes).

Experiment 2. The exception from the general rule that higher weights usually do not cause high degradations of solution quality is the set of instances of the Eight-Puzzle (it is contained in the HSP2 distribution): the average loss in quality went up to 70% when applying weights larger then 10.0 (see Fig. B.3(b)). For the random puzzle *prob04*, the number of generated and expanded nodes more than doubles when the weight is increased from 3.0 to 6.0 (but both numbers fall to less than half of their original value when a weight of 11.0 is applied). The harder instances *prob02* and *prob03* show a similar blow up for the weights 10.0 and 11.0.

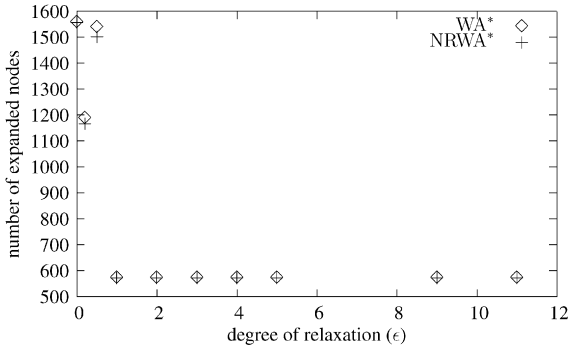
Experiment 3. A potential negative effect of higher weights is the increase in the number of reopenings, since the inconsistency of the weighted heuristic increases. We observed that this effect diminishes for several domains (e.g. Blocksworld, Depots and the Logistics domain), if the weights are further increased. E.g., the problems of the Depots domain have reopenings for a weight $w = 1.20$, but not for $w \geqslant 2.0$, and those of the Logistics domain have reopenings for $1.10 \leqslant w \leqslant 2.0$, but not for $w \geqslant 3.0$). Here the number of reopenings goes back to zero again since the positive effects of the relaxation (i.e., a more focused search that helps good solutions more quickly) take over. The benefit from the relaxation becomes dominating for the higher weights. For the sliding tile puzzle, it is vice versa (that is, the reduction in run time resulting from the relaxation is absorbed more and more by the increase in reopenings). Nevertheless, the total number of generated and expanded nodes for the examined set of Eight-Puzzles is not increased significantly for higher weights. This is because the weights with an aforementioned negative effect are different for distinct problems (and also have positive effects on other problems). So for this experiment, WA* worked well.

Experiment 4. The first example for poor performance of WA* is in the Driverlog domain from IPC3, where one problem of size 2 (named *pfile1*) is completely insensitive to changes in the weight. Regardless of the applied weight, the same number of nodes were created and expanded, yielding the same solution as with A*. The second example is the set of problems of size 2 in the Freecell domain from the AIPS2000 benchmark collection: A* was able to solve them all (but none of the bigger problems) on our machine within 100 seconds. Unfortunately, neither WA* nor the non-reopening variant can solve these problems more quickly: for the weights $1.01, 1.02, \ldots, 1.10$ there is no change in the number of expanded or generated nodes compared to that of A*. For the weight 1.20 the number of expanded or generated nodes, and also the run time already more than doubles, and the higher weights 2.0, 5.0, and 10.0 result in a dramatic increase of generated and created nodes. Hence the run time limit of 100 seconds is exceeded. Hansen and Zhou state that weighted A* works best when many close-to-optimal solutions are available [38]. Freecell in fact is an example of a domain with a rather sparse solution space which probably is the reason for the observed behavior.

The same authors also report a successful experiment applying AWA* to a Freecell problem of size 3, using a weight of 2.0. The algorithm then failed to find a solution before running out of memory with higher weights of 5.0 and 10.0. We applied WA* and NRWA* on the problem of the IPC3 suite of size 3. Our results are consistent with the observations of Hansen and Zhou, as with a weight of 2.0 a solution is found in less than two seconds with both variants of weighted A*. Moreover, both methods result in the same solution length. The non-reopening variant NRWA* performs significantly faster as only 2879 nodes are expanded (and 15118 nodes are generated) instead of 3387 nodes expanded (and 17757 generated) by WA*. Here, many reopenings are avoided by the use of the non-reopening variant. Both variants perform much worse

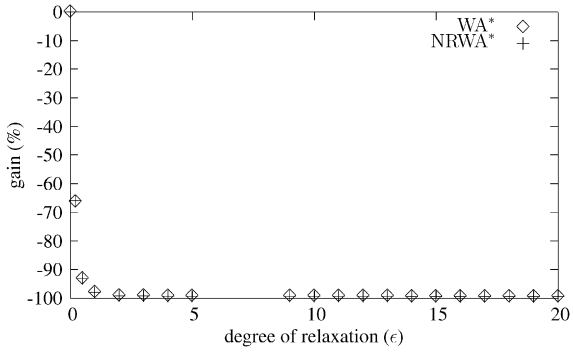


(a) Node expansions for Depots domain.

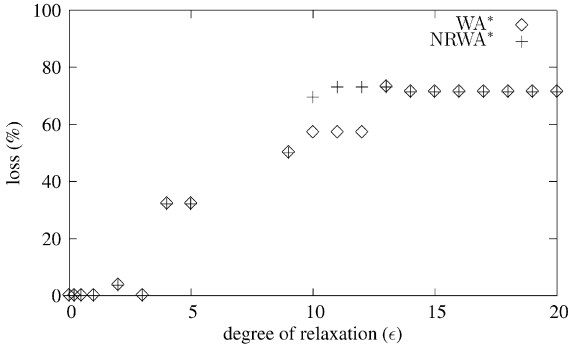


(b) Node expansions for Satellite domain.

Fig. B.2. Total number of expansions for selected domains.



(a) Gain for Puzzle domain.



(b) Loss for Puzzle domain.

Fig. B.3. Gain and loss for the Puzzle domain.

for a weight of 1.50 and fail completely for all weights greater than or equal to 3.0 that have been tried (5.0, 10.0, 20.0, and 50.0).

Experiment 5. The (only) examples where NRWA* yields a higher run time than WA* are in the Depots domain from IPC3 where the weight 1.20 (and, less dramatically, also the weight 1.50) causes a high increase in the number of generated and expanded nodes (see Fig. B.2(a) for the expanded nodes).

References

[1] S. Akers, Functional testing with binary decision diagrams, in: Eighth Annual Conf. on Fault-Tolerant Computing, 1978, pp. 75–82.
[2] P. Bertoli, A. Cimatti, M. Roveri, Heuristic search symbolic model checking = efficient conformant planning, in: Proc. of the Int'l Joint Conf. on Artificial Intelligence, 2001, pp. 467–472.
[3] P. Bertoli, A. Cimatti, M. Roveri, P. Traverso, Planning in non-deterministic domains under partial observability via symbolic model checking, in: Proc. of the Int'l Joint Conf. on Artificial Intelligence, 2001, pp. 473–478.
[4] B. Bollig, M. Löbbing, I. Wegener, Simulated annealing to improve variable orderings for OBDDs, in: Int'l Workshop on Logic Synth., 1995, pp. 5b:5.1–5.10.
[5] B. Bollig, I. Wegener, Improving the variable ordering of OBDDs is NP-complete, IEEE Trans. on Comp. 45 (9) (1996) 993–1002.
[6] B. Bonet, H. Geffner, Heuristic search planner 2.0, AI Magazine 22 (3) (2001) 77–80.
[7] B. Bonet, H. Geffner, Planning as heuristic search, Artificial Intelligence 129 (1) (2001) 5–33.
[8] K. Brace, R. Rudell, R. Bryant, Efficient implementation of a BDD package, in: Design Automation Conf., 1990, pp. 40–45.
[9] R.E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Trans. on Comp. 35 (8) (1986) 677–691.
[10] P. Buch, A. Narayan, A. Newton, A. Sangiovanni-Vincentelli, Logic synthesis for large pass transistor circuits, in: Int'l Conf. on CAD, 1997, pp. 663–670.
[11] T. Bylander, Complexity results for planning, in: Proc. of the Int'l Joint Conference on Artificial Intelligence, 1991, pp. 274–279.
[12] T. Cain, Practical optimisation for A^* path generation, in: AI Game Programming Wisdom, Charles River Media, 2002, pp. 146–152.
[13] P. Chakrabarti, S. Ghose, A. Acharya, S. de Sarkar, Heuristic search in restricted memory, Artificial Intelligence 47 (1989) 197–221.
[14] S. Chang, M. Marek-Sadowska, T. Hwang, Technology mapping for TLU FPGAs based on decomposition of binary decision diagrams, IEEE Trans. on CAD 15 (10) (1996) 1226–1236.
[15] M. Chrzanowska-Jeske, Z. Wang, Mapping of symmetric and partially-symmetric functions to the CA-type FPGAs, in: Proc. Midwest Symp. on Circuits and Systems, 1995, pp. 290–293.
[16] M. Chrzanowska-Jeske, Z. Wang, Y. Xu, A regular representation for mapping to fine-grain locally-connected FPGAs, in: Proc. Int'l Symp. on Circuits and Systems, 1997, pp. 2749–2752.
[17] A. Cimatti, E. Giunchiglia, F. Giunchiglia, P. Traverso, Planning via model checking: A decision procedure for AR, in: Proc. of the European Conf. on Planning, 1997, pp. 130–142.
[18] A. Cimatti, M. Roveri, P. Traverso, Automatic OBDD-based generation of universal plans in non-deterministic domains, in: Proc. of the National Conf. on Artificial Intelligence, 2000, pp. 875–881.
[19] Collaborative Benchmarking Laboratory, 1993 LGSynth Benchmarks, North Carolina State University, Department of Computer Science, 1993.
[20] R. Dechter, J. Pearl, Generalized best-first search strategies and the optimality of A^*, Journal of the Association for Computing Machinery 32 (3) (1985) 505–536.
[21] R.B. Dial, Algorithm 360: Shortest path forest with topological ordering, Commun. ACM 12 (11) (1969) 632–633.
[22] E.W. Dijkstra, A note on two problems in connexion with graphs, Numerische Mathematik 1 (1959) 269–271.
[23] R. Drechsler, B. Becker, N. Göckel, A genetic algorithm for variable ordering of OBDDs, IEE Proc. Comp. Digital Techniques 143 (6) (1996) 364–368.
[24] R. Drechsler, N. Drechsler, W. Günther, Fast exact minimization of BDDs, IEEE Trans. on CAD 19 (3) (2000) 384–389.
[25] R. Drechsler, W. Günther, F. Somenzi, Using lower bounds during dynamic BDD minimization, IEEE Trans. on CAD 20 (1) (2001) 51–57.
[26] R. Ebendt, R. Drechsler, The effect of lower bounds in dynamic BDD reordering, IEEE Trans. on CAD 25 (5) (2006) 902–909.
[27] R. Ebendt, W. Günther, R. Drechsler, An improved branch and bound algorithm for exact BDD minimization, IEEE Trans. on CAD 22 (12) (2003) 1657–1663.
[28] R. Ebendt, W. Günther, R. Drechsler, Combining ordered best-first search with branch and bound for exact BDD minimization, IEEE Trans. on CAD 24 (10) (2005) 1515–1529.
[29] S. Edelkamp, T. Mehler, Byte code distance heuristics and trail direction for model checking Java programs, in: Proc. of the Workshop on Model Checking and Artificial Intelligence, 2003, pp. 69–76.
[30] S. Edelkamp, F. Reffel, OBDDs in heuristic search, in: Advances in Artificial Intelligence, in: LNAI, Springer Verlag, 1998, pp. 81–92.
[31] J. Ellson, E. Gansner, G. Low, D. Dobkin, E. Koutsofios, S. North, K.-P. Vo, G. Woodhull, Graphviz – graph visualization software, Website http://www.graphviz.org, 2000–2008.
[32] S. Ercolani, G.D. Micheli, Technology mapping for electronically programmable gate arrays, in: Proc. of the 28th Design Automation Conf., 1991, pp. 234–239.
[33] A. Felner, S. Kraus, R. Korf, KBFS: K-best first search, Annals of Mathematics and Artificial Intelligence 39 (1–2) (2003) 19–39.
[34] Z. Feng, E. Hansen, S. Zilberstein, Symbolic generalization for on-line planning, in: Proc. of the Annual Conf. on Uncertainty in Artificial Intelligence, 2003, pp. 209–216.
[35] F. Ferrandi, A. Macii, E. Macii, M. Poncino, R. Scarsi, F. Somenzi, Symbolic algorithms for layout-oriented synthesis of pass transistor logic circuits, in: Int'l Conf. on CAD, 1998, pp. 235–241.
[36] S. Friedman, K. Supowit, Finding the optimal variable ordering for binary decision diagrams, IEEE Trans. on Comp. 39 (5) (1990) 710–713.
[37] M. Garey, D. Johnson, Computers and Intractability. A Guide to the Theory of NP-Completeness, W.H. Freeman, New York, 1979.
[38] E. Hansen, R. Zhou, Anytime heuristic search, Journal of Artificial Intelligence Research 28 (2007) 267–297.
[39] E. Hansen, R. Zhou, Z. Feng, Symbolic heuristic search using decision diagrams, in: Symp. on Abstraction, Reformulation and Approximation, 2002, pp. 83–98.
[40] P. Hart, N. Nilsson, B. Raphael, A formal basis for the heuristic determination of minimum cost paths, IEEE Trans. Syst. Sci. Cybern. 2 (1968) 100–107.
[41] P. Haslum, H. Geffner, Admissible heuristics for optimal planning, in: Proc. Int'l Conf. on AI Planning Systems, 2000, pp. 70–82.
[42] N. Ishiura, H. Sawada, S. Yajima, Minimization of binary decision diagrams based on exchange of variables, in: Int'l Conf. on CAD, 1991, pp. 472–475.
[43] R. Jensen, R. Bryant, M. Veloso, SetA*: An efficient BDD-based A^* algorithm, in: Proc. of the National Conf. on Artificial Intelligence, 2002, pp. 668–673.
[44] R. Jensen, E. Hansen, S. Richards, R. Zhou, Memory-efficient symbolic heuristic search, in: Proc. of Int'l Conf. on Automated Planning and Scheduling, 2006, pp. 304–313.
[45] R. Jensen, M. Veloso, ASET: A multi-agent planning language with nondeterministic durative tasks for BDD-based fault tolerant planning, in: Proc. of Int'l Conf. on Automated Planning and Scheduling: Workshop on Multiagent Planning and Scheduling, 2005, pp. 58–65.

[46] S.-W. Jeong, T.-S. Kim, F. Somenzi, An efficient method for optimal BDD ordering computation, in: Int'l Conf. on VLSI and CAD, 1993.
[47] D.B. Johnson, Efficient algorithms for shortest paths in sparse networks, J. ACM 24 (1) (1977) 1–13.
[48] H. Kaindl, G. Kainz, Bidirectional heuristic search reconsidered, Journal of Artificial Intelligence Research 7 (1997) 283–317.
[49] H. Kaindl, A. Khorsand, Memory-bounded bidirectional search, in: Proc. of the 12th National Conf. on Artificial Intelligence, 1994, pp. 1359–1364.
[50] H. Kautz, B. Selman, Unifying SAT-based and graph-based planning, in: Proc. Int. Joint Conf. on Artificial Intelligence, 1999, pp. 318–327.
[51] H. Kobayashi, H. Imai, Improvement of the A^* algorithm for multiple sequence alignment, in: Proc. of the 9th Workshop on Genome Informatics, 1998, pp. 120–130.
[52] A. Köll, H. Kaindl, A new approach to dynamic weighting, in: Proc. of the European Conf. on Artificial Intelligence, 1992, pp. 16–17.
[53] A. Köll, H. Kaindl, Bidirectional best-first search with bounded error: Summary of results, in: Proc. of the 13th International Joint Conf. on Artificial Intelligence, 1993, pp. 217–223.
[54] R.E. Korf, Planning as search: A quantitative approach, Artificial Intelligence 33 (1) (1987) 65–68.
[55] R.E. Korf, Linear-space best-first search, Artificial Intelligence 62 (1) (1993) 41–78.
[56] R.E. Korf, A complete anytime algorithm for number partitioning, Artificial Intelligence 106 (2) (1998) 181–203.
[57] R.E. Korf, An improved algorithm for optimal bin packing, in: Proc. of Int'l Joint Conf. on Artificial Intelligence, 2003, pp. 1252–1258.
[58] R.E. Korf, Optimal rectangle packing: New results, in: Proc. of Int'l Conf. on Automated Planning and Scheduling, 2004, pp. 142–149.
[59] R.E. Korf, W. Zhang, Divide-and-conquer frontier search applied to optimal sequence alignment, in: Proc. of the 17th National Conf. on Artificial Intelligence, 2000, pp. 910–916.
[60] R.E. Korf, W. Zhang, I. Thayer, H. Hohwald, Frontier search, J. ACM 52 (5) (2005) 715–748.
[61] K. Kotecha, N. Gambhava, A hybrid genetic algorithm for minimum vertex cover problem in: Proc. of the Indian Int'l Conf. on Artificial Intelligence, 2003, pp. 904–913.
[62] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, S. Thrun, Anytime search in dynamic graphs, Artificial Intelligence 172 (14) (2008) 1613–1643.
[63] M. Likhachev, G. Gordon, S. Thrun, ARA*: Formal analysis, Technical report of the Carnegie Mellon University, 2003.
[64] M. Likhachev, G.J. Gordon, S. Thrun, ARA*: Anytime A^* with provable bounds on sub-optimality, in: S. Thrun, L. Saul, B. Schölkopf (Eds.), Advances in Neural Information Processing Systems, vol. 16, MIT Press, Cambridge, MA, 2004.
[65] D. Long, M. Fox, The efficient implementation of the plan-graph in STAN, J. of Artificial Intelligence Research 10 (1999) 85–115.
[66] L. Macchiarulo, L. Benini, E. Macii, On-the-fly layout generation for PTL macrocells, in: Design, Automation and Test in Europe, 2001, pp. 546–551.
[67] A. Mukherjee, R. Sudhakar, M. Marek-Sadowska, S. Long, Wave steering in YADDs: A novel, non-iterative synthesis and layout technique, in: Design Automation Conf., 1999, pp. 466–471.
[68] R. Murgai, Y. Nishizaki, N. Shenoy, R. Brayton, A. Sangiovanni-Vincentelli, Logic synthesis for programmable gate arrays, in: Proc. of the 27th Design Automation Conf., 1991, pp. 620–625.
[69] N. Nilsson, Principles of Artificial Intelligence, Tioga Publishing Company, Palo Alto, CA, 1980.
[70] J. Pearl, J. Kim, Studies in semi-admissible heuristics, IEEE Trans. on Pattern Analysis and Machine Intelligence PAMI-4 (4) (1982) 392–399.
[71] I. Pohl, Heuristic search viewed as path finding in a graph, Artificial Intelligence 1 (3) (1970) 193–204.
[72] I. Pohl, The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving, in: Proc. of the 3rd Int. Joint Conf. on Artificial Intelligence, 1973, pp. 12–17.
[73] K. Qian, A. Nymeier, Heuristic search algorithms based on symbolic data structures, in: Proc. of the Australian Conf. on Artificial Intelligence, 2003, pp. 966–979.
[74] D. Ratner, M. Warmuth, Finding a shortest solution for the $(n \times n)$-extension of the 15-puzzle is intractable, J. Symbolic Computation 10 (2) (1990) 111–137.
[75] F. Reffel, S. Edelkamp, Error detection with directed symbolic model checking, in: World Congress on Formal Methods, 1999, pp. 195–211.
[76] R. Rudell, Dynamic variable ordering for ordered binary decision diagrams, in: Int'l Conf. on CAD, 1993, pp. 42–47.
[77] S. Russell, Efficient memory-bounded search methods, in: Proc. of the 10th European Conf. on Artificial Intelligence, vol. 16, 1992, pp. 701–710.
[78] S. Schroedl, An improved search algorithm for optimal multiple-sequence alignment, Journal of Artificial Intelligence Research 23 (2005) 587–623.
[79] C. Shannon, A symbolic analysis of relay and switching circuits, Trans. AIEE 57 (1938) 713–723.
[80] R. Shelar, S. Sapatnekar, Pass Transistor Logic Synthesizer Version 1.0 (PTLS), University of Michigan, 2002.
[81] D. Sieling, Nonapproximability of OBDD minimization, Information and Computation 172 (2) (2002) 103–138.
[82] D. Sieling, I. Wegener, Reduction of BDDs in linear time, Information Processing Letters 48 (3) (1993) 139–144.
[83] F. Somenzi, CU decision diagram package release 2.4.1, University of Colorado at Boulder, available at http://vlsi.colorado.edu/~fabio/CUDD/, 2004.
[84] I. Wegener, Worst case examples for operations on OBDDs, Information Processing Letters 74 (2000) 91–96.
[85] D. Weld, An introduction to least commitment planning, AI Magazine 15 (4) (1994) 27–61.
[86] C. Yang, M. Ciesielski, BDS: A BDD-based logic optimization system, IEEE Trans. on CAD 21 (7) (2002) 866–876.
[87] R. Zhou, E. Hansen, Memory-bounded A^* graph search, in: 15th Int. Florida Artificial Intelligence Research Soc. Conf., 2002, pp. 203–209.
[88] R. Zhou, E. Hansen, Multiple sequence alignment using anytime A^*, in: Proc. of the 18th National Conf. on Artificial Intelligence, Student Abstract, 2002, pp. 975–976.
[89] R. Zhou, E. Hansen, Sparse-memory graph search, in: Proc. of the 18th Joint Conf. on Artificial Intelligence, 2003, pp. 1259–1266.
[90] R. Zhou, E. Hansen, Sweep A^*: Space-efficient heuristic search in partially ordered graphs, in: Proc. of the 15th IEEE Int. Conf. on Tools with Artificial Intelligence, 2003, pp. 427–434.