View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Elsevier - Publisher Connector

The Journal of Logic and Algebraic Programming 81 (2012) 46–69

Contents lists available at SciVerse ScienceDirect

The Journal of Logic and Algebraic Programming

journal homepage: www.elsevier.com/locate/jlap

Extended beam search for non-exhaustive state space analysis

A.J. Wijs [a,*], M. Torabi Dashti [b]

[a] Eindhoven University of Technology, Den Dolech 2, 5612 AZ Eindhoven, The Netherlands
[b] ETH Zürich, CNB F 109, Universitätstrasse 6, 8092 Zürich, Switzerland

ARTICLE INFO

Article history:
Available online 5 September 2011

Keywords:
Model checking
Formal analysis
State space generation
Heuristics
Guided traversal

ABSTRACT

State space explosion is a major problem in both qualitative and quantitative model checking. This article focuses on using beam search, a heuristic search algorithm, for pruning weighted state spaces while generating. The original beam search is adapted to the state space generation setting and two new variants, motivated by practical case studies, are devised. These beam searches have been implemented in the μCRL toolset and applied on several case studies reported in the article.

© 2011 Elsevier Inc. All rights reserved.

1. Introduction

State space explosion is a major problem in model checking. To mitigate this problem, over the years a number of techniques have emerged to prune, or postpone generating, parts of the state space that are not, or do not seem, essential, given the verification task at hand. Prominent examples of pruning and postponing techniques are partial order reduction (Por) [13,49], and directed model checking (Dmc) [5,19], respectively. Intuitively, Por algorithms guarantee that no essential information is lost due to pruning. Dmc algorithms, however, use heuristics to guide the generation, so that the states which are most relevant to the verification task are generated first. Both Dmc and Por are in line with the core idea of model checking when studying qualitative properties, i.e. either to exhaustively search the complete state space to find any corner case bug, or guarantee that the pruned states are immaterial for the verification task.

This article, in contrast, focuses mainly on heuristic pruning methods which are applicable to verification of quantitative properties of systems [11], e.g. finding an optimal schedule for an industrial batch processor, calculating the probability of a frame loss for a network device, and determining the minimal time needed to prepare a product on an assembly line. When using model checkers for quantitative analyses of systems often (1) solutions (represented by paths from an initial state to a so-called *goal state*) to the problem at hand densely populate the state space, and (2) near-optimal answers are sufficiently acceptable in practice. Remark that these two features are not common in qualitative analysis problems, where goal states usually denote corner case bugs, and are therefore sparsely present in the state space, and moreover, definitive answers are needed. Therefore, quantitative analyses allow pruning techniques which are not necessarily useful for qualitative analyses.

In particular, we investigate how *beam search* (Bs) can be used for weighted state space generation, to solve quantitative problems. Bs is a heuristic search method for combinatorial optimisation problems, which has extensively been studied in artificial intelligence and operations research, among others by [42,55,26,61,58,15]. Conceptually, Bs is similar to breadth-first search (Bfs), as it progresses level by level through a highly structured search tree containing all possible solutions to a problem. Bs, however, does not explore all the encountered nodes. At each level, all the nodes are evaluated using a heuristic cost (or priority) function, but only a fixed number of them is selected for further examination. This aggressive pruning

* Corresponding author.
E-mail addresses: A.J.Wijs@tue.nl (A.J. Wijs), mohammad.torabi@inf.ethz.ch (M. Torabi Dashti).

1567-8326/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jlap.2011.06.002

heavily decreases the generation time and memory consumption, but may in general miss essential parts of the tree for the problem at hand, since wrong decisions can be made while pruning. Therefore, Bs has so far been mainly used in search trees with a high density of goal nodes. Scheduling tasks, for instance, have been perfect targets for using Bs: the goal is to optimally schedule a certain number of jobs and resources, while near-optimal schedules, which densely populate the tree, are in practice acceptable.

Using Bs in state space generation is an attempt towards integrating functional analysis, to which state spaces are usually subjected, and quantitative analysis. In the current article, we motivate and thoroughly discuss adapting two Bs techniques called *detailed* (Dbs) and *priority* (Pbs) Bs to deal with arbitrary structures of state spaces. We remark that model checkers (e.g. Spin [34], Uppaal [4], the μCRL toolset [8], the mCRL2 toolset [31], the LTSmin toolset [9], and the Cadp toolbox [27]) usually have very expressive input languages. Therefore, Bs must be adapted to deal with more general state spaces, compared to simple search trees to which Bs has been traditionally applied. Next, we extend the classic Bss in two directions. First, we propose *flexible* Bs, which, broadly speaking, does not stick to a fixed number of states to be selected at each search level. This partially mitigates the problem of determining this exact fixed number in advance. Second, we introduce the notion of *synchronised* Bs, which aims at separating the heuristic pruning phase from the underlying exploration strategy.

We have implemented the aforementioned variants in the μCRL [32] state space generation toolset [8]. The process algebraic language μCRL, an extension of ACP with abstract data types, comes with stable and mature tool support. A μCRL specification consists of data type declarations and process behaviour definitions, where processes and actions can be parametrised with data. Data are typed in μCRL and types can have recursive definitions. Each non-empty data type has constructors and possibly non-constructors associated to it. The semantics of non-constructors is given by means of equations. The specification of a component is a guarded recursive equation that is constructed from a finite set of action labels, process algebraic operators and recursion variables. The μCRL toolset can be used for automatically generating state spaces from μCRL specifications. The Cadp toolbox [27] can then be used for verifying regular alternation-free μ-calculus properties of the resulting state spaces. Augmenting the μCRL toolset with Bs therefore enables us to perform advanced qualitative model checking in the same framework where we perform our quantitative analysis.

Experimental results for several scheduling case studies are reported. It should be stressed that, even though this article focusses on applying Bs to solve scheduling problems, the techniques described here are also applicable for other forms of quantitative model checking, such as real-time and stochastic model checking. We briefly discuss this in Section 10.3.
Road map: First, in Section 2, we present the basic notions necessary for understanding this article. We describe the general search algorithm called *best-first search* and present a specific instance called *uniform-cost search*. After that, we propose a new extension of best-first search, called *multi-phase* best-first search, in Section 3. Two classic Bss for highly structured trees are described in Section 4; here, we temporarily deviate from the model checking setting, and present Bs in its 'traditional' setting. Section 5 deals with the adaptation of two existing variants of Bs to the state space generation setting. There we also propose our extensions to the Bs algorithms. After that we focus in Section 6 on the implementation of some of these adapted and extended Bs algorithms in the μCRL toolset. Related issues such as memory management and selecting heuristic functions are discussed in Sections 7 and 8, respectively. After that, experimental results for several scheduling problems are discussed in Section 9. Section 10 presents our related work, including other uses of Bs, connections with other search algorithms, and other possible application areas for Bs. Finally, Section 11 concludes the article.

2. Searching through weighted state spaces

Definition 1 (*Weighted state space*)**.** A *weighted state space* or *weighted labelled transition system (*Wlts*)* is a quintuple $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, where $\mathscr{S}$ is a set of states, $\mathscr{I} \subseteq \mathscr{S}$ is a set of initial states, $\mathscr{A}$ is a finite set of action labels, $\mathscr{C} : \mathscr{A} \rightarrow \mathbb{K}$, with $\mathbb{K}$ a cost domain, is a total function assigning costs to action labels, and $\mathscr{T} \subseteq \mathscr{S} \times \mathscr{A} \times \mathscr{S}$ is the transition relation.

A state s' is called *reachable* from state s iff $s \rightarrow^* s'$, where $\rightarrow^*$ is the reflexive transitive closure of $\xrightarrow{\ell}$ for any $\ell \in \mathscr{A}$. When checking a *reachability* property, one searches for an $s \in \mathscr{G}$, where $\mathscr{G} \subseteq \mathscr{S}$ is a given set of goal states, such that there exists an $s' \in \mathscr{I}$ for which $s' \rightarrow^* s$.

The set of enabled transitions in state s of Wlts $\mathscr{M}$ is defined as $en_{\mathscr{M}}(s) = \{t \in \mathscr{T} \mid \exists s' \in \mathscr{S},\ \ell \in \mathscr{A}.\, t = s \xrightarrow{\ell} s'\}$. For $T \subseteq \mathscr{T}$, we define $nxt_{\mathscr{M}}(s, T) = \{s' \in \mathscr{S} \mid \exists \ell \in \mathscr{A}.\, s \xrightarrow{\ell} s' \in T\}$. Therefore, $nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$ is the set of successor states of s. Whenever $en_{\mathscr{M}}(s) = \emptyset$, we call s a *deadlock* state. Finally, in the state space setting, an action may have several data parameters, which are considered to be included in the action label ℓ. These parameters may be defined over infinite domains, but as we require $\mathscr{A}$ to be finite, the sets of concrete values for the parameters, as they appear in $\mathscr{M}$, should be finite as well. Whenever we compare action labels, for instance $\ell = a$, we ignore the parameters. We are aware of this discrepancy, but trying to avoid it would lead to unnecessary complications.

Given a cost domain $\mathbb{K}$, in a Wlts, every action in $\mathscr{A}$ is associated with a cost $c \in \mathbb{K}$. Such a totally ordered cost domain can be a subset of any known set of numbers, e.g. $\mathbb{K} \subseteq \mathbb{N}$ or $\mathbb{K} \subseteq \mathbb{R}$. We do not consider negative cost values here. Note that a standard Lts can be seen as a Wlts where all $\ell \in \mathscr{A}$ have the same associated cost.

WLTSs can typically be used to deal with priced problems, such as scheduling or planning problems. These can be modelled as reachability problems, as shown by e.g. [5,57,67], where on top of the usual question whether a goal state $s \in \mathscr{G}$ can be reached or not, it is desired to find a trace to such a goal state with minimal cumulated cost. A first attempt at defining the *cumulated cost* of a state s follows:

Definition 2 (*('Informal') Cumulated cost*)**.** Given a WLTS $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, we say that the *cumulated cost* of state $s \in \mathscr{S}$, denoted $g(s)$ (with $g : \mathscr{S} \to \mathbb{K}$), equals $g(s') + c$ with $s' \in \mathscr{S}$ iff $\exists \ell \in \mathscr{A}. s' \xrightarrow{\ell} s$ and $\mathscr{C}(\ell) = c$. For $s \in \mathscr{I}$, $g(s) = 0$.

One problem with this definition becomes readily apparent, namely that for any $s \in \mathscr{S}$, $g(s)$ need not be unique, since many traces may lead from $\mathscr{I}$ to s. More on this later, but it raises the issue of *re-opening* of states in DMC. Typically, when generating a WLTS, and determining $g(s)$ for a state s on-the-fly, one may discover smaller $g(s)$ along the way. Because of this, there is a need to refer to the *minimal weighted distance* from a set of states S to a set of states S'. First, we write a trace through $\mathscr{M}$ from a state $s \in \mathscr{S}$ to a state $s' \in \mathscr{S}$ as a sequence of transitions $\sigma_{\mathscr{M}}(s, s') = \langle s_0 \xrightarrow{\ell_0} s_1, s_1 \xrightarrow{\ell_1} s_2, \ldots, s_{n-1} \xrightarrow{\ell_{n-1}} s_n \rangle$, with $n > 0$, $s = s_0$, $s' = s_n$, and $(s_i \xrightarrow{\ell_i} s_{i+1}) \in \mathscr{T}$ for all $0 \leq i < n$. Furthermore, we call the set of all possible traces between two states $s, s' \in \mathscr{S}$ in $\mathscr{M}$ the set $\Sigma_{\mathscr{M}}(s, s') = \{\langle s_0 \xrightarrow{\ell_0} s_1, s_1 \xrightarrow{\ell_1} s_2, \ldots, s_{n-1} \xrightarrow{\ell_{n-1}} s_n \rangle | s = s_0 \wedge s' = s_n \wedge n > 0 \wedge \forall i < n. (s_i \xrightarrow{\ell_i} s_{i+1}) \in \mathscr{T}\}$. Next, we can define the notion of minimal weighted distance between (sets of) states:

Definition 3 (*Minimal weighted distance between (sets of) states*)**.** Given a WLTS $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$ and two states $s, s' \in \mathscr{S}$. For a given trace $\sigma_{\mathscr{M}}(s, s') = \langle s \xrightarrow{\ell_0} s_0, s_0 \xrightarrow{\ell_1} s_1, \ldots, s_{n-1} \xrightarrow{\ell_n} s' \rangle$, with $n \geq 0$, we define its weighted distance $d(\sigma_{\mathscr{M}}(s, s'))$ as $\sum_{0 \leq i \leq n} \mathscr{C}(\ell_i)$. Furthermore, we say that the *minimal weighted distance between two states* $s, s' \in \mathscr{S}$, denoted $\delta_{\mathscr{M}}(s, s')$, equals $min(\{d(\sigma) | \sigma \in \Sigma_{\mathscr{M}}(s, s')\})$, and that the *minimal weighted distance between two sets of states* $S, S' \subseteq \mathscr{S}$, denoted $\Delta_{\mathscr{M}}(S, S')$, equals $min(\{\delta_{\mathscr{M}}(s, s') | s \in S \wedge s' \in S'\})$. In case s' is not reachable from s, we say that $\delta_{\mathscr{M}}(s, s') = \infty$, and if S' is not reachable from S, meaning that there are no $s \in S, s' \in S'$ with $s \to^* s'$, then $\Delta_{\mathscr{M}}(S, S') = \infty$.

Clearly, if $S \cap S' \neq \emptyset$ then $\Delta_{\mathscr{M}}(S, S') = 0$, since there is a state s for which $s \in S$ and $s \in S'$, which has a trace of length 0 to itself.

Now, we return to the notion of cumulated cost. As noted, $g(s)$, as defined in Definition 2, is in fact a multivalued function, since $g(s)$ may have several values, in case there are multiple traces leading from $\mathscr{I}$ to s. We observe, however, that in on-the-fly searching, g is not merely a multivalued function, but a (partial) function which is at times redefined, namely each time a state is (re-)opened; at any particular moment during a search, g is a partial function. Explicitly taking on-the-fly searching into account, we finally define the notion of cumulated cost in Definition 4. There, $s \to_T^* s'$ denotes that s' is reachable from s *through the set of transitions* T, i.e. there are $s_0, \ldots, s_n \in \mathscr{S}$ and $\ell_0, \ldots, \ell_{n+1} \in \mathscr{A}$, with $n \geq 0$, such that $s \xrightarrow{\ell_0} s_0 \in T$, $s_i \xrightarrow{\ell_{i+1}} s_{i+1} \in T$ for $0 \leq i \leq n-1$, and $s_n \xrightarrow{\ell_{n+1}} s' \in T$. Fig. 1 shows an example of (re-)defining g on-the-fly. We return to this figure later on. In Definition 4, we use the triple set notation $R = (X, Y, F)$ to define a binary relation R (note that a partial function is a specific kind of binary relation), with X the domain, Y the codomain, and F a set of pairs (x, y) defining that $R(x) = y$. Furthermore, we use a union operator $\sqcup$ on binary relations, which is defined as $(X, Y, F) \sqcup (X, Y, F') = (X, Y, F \cup F')$, and 'undefined' is represented by '$\perp$', i.e. if R is undefined for s, we write $R(s) = \perp$.

Definition 4 (*Cumulated cost*)**.** Given a WLTS $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, we recursively define the *cumulated cost* function relative to a given set of transitions T (called the *scope*) $g_T : \mathscr{S} \to \mathbb{K}$ as follows:

- $g_\emptyset = (\mathscr{S}, \mathbb{K}, \{\langle s, 0 \rangle | s \in \mathscr{I}\})$;
- Given a (partial) function g_T and a transition $(s_0 \xrightarrow{\ell} s_1) \notin T$ such that $g_T(s_0)$ is defined. Then, we have

$$g_{T \cup \{s_0 \xrightarrow{\ell} s_1\}} = mc(g_T \sqcup (\mathscr{S}, \mathbb{K}, \{\langle s_1, g_T(s_0) + \mathscr{C}(\ell) \rangle\}) \sqcup R_T(s_1, g_T(s_0) + \mathscr{C}(\ell))),$$

with $R_T(s_1, g) =$

$$\begin{cases} (\mathscr{S}, \mathbb{K}, \emptyset), & \text{if } g \geq g_T(s_1) \vee en_{\mathscr{M}}(s_1) \cap T = \emptyset \\ \bigsqcup_{(s_1, \ell, s') \in en_{\mathscr{M}}(s_1) \cap T} ((\mathscr{S}, \mathbb{K}, \{\langle s', g + \mathscr{C}(\ell) \rangle\}) \sqcup R_T(s', g + \mathscr{C}(\ell))), & \text{otherwise} \end{cases}$$

and $mc((X, Y, F)) = (X, Y, \{\langle s, g \rangle \in F | \neg \exists g' < g. \langle s, g' \rangle \in F\})$.

Definition 4 explains how the cumulated cost function g can be constructed on-the-fly; initially we have $g_\emptyset$, which is defined only for $\mathscr{I}$, such that for all $s \in \mathscr{I}$, $g_\emptyset(s) = 0$. As the search scope is increased, the cumulated cost function is redefined, which is made explicit here by increasing the scope T. When adding a transition $s_0 \xrightarrow{\ell} s_1$ to the scope T, such

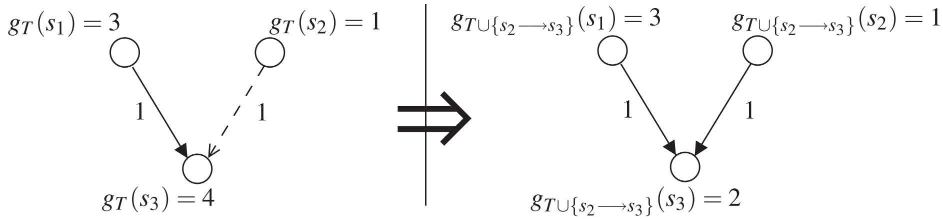

Fig. 1. Monotonicity example.

that s_0 is reachable in T from $\mathscr{I}$ (i.e. there exists an $s \in \mathscr{I}$ such that $s \rightarrow_T^* s_0$), the new partial function $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}$ is equal to the partial function g_T extended with the definitions $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}(s_1) = g_T(s_0) + \mathscr{C}(\ell)$, which associates a new cumulated cost with s_1, and $R_T(s_1, g_T(s_0) + \mathscr{C}(\ell))$. Binary relation R_T provides new cumulated costs for all states reachable from s_1 in T, and represents re-opening s_1. Note in the definition of R_T that if s_1 has already been (partially) explored, $en_{\mathscr{M}}(s_1) \cap T \neq \emptyset$. If s_1 now gets a lower cumulated cost than in an earlier visit, i.e. $g_T(s_0) + \mathscr{C}(\ell) < g_T(s_1)$, then we need to recompute the cumulated costs for the successors in T of s_1 as well. Since T is of finite size, the recursive computation of R_T is guaranteed to terminate.

Finally, mc strips a set of $\langle$state, cost$\rangle$-pairs of all non-minimal costs, i.e. for each state, only the minimal known cost is kept. This means that a given binary relation is converted to a partial function. Note that if $g_T(s_1) \leq g_T(s_0) + \mathscr{C}(\ell)$, computation of $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}}$ can be simplified to $mc(g_T) = g_T$. In case $g_T(s_1) = \perp$, we have $g_{T\cup\{s_0 \xrightarrow{\ell} s_1\}} = mc(g_T \sqcup (\mathscr{S}, \mathbb{K}, \{\langle s_1, g_T(s_0) + \mathscr{C}(\ell)\rangle\}))$, since s_1 is a newly visited state, and hence has no outgoing transitions in T.

It follows that for all $T \subset \mathscr{T}$, $g_T : \mathscr{S} \rightarrow \mathscr{S}$ is a partial function. In fact, the only total function is $g_{\mathscr{T}}$, under the assumption that all states $s \in \mathscr{S}$ are reachable from $\mathscr{I}$. As we continue searching a WLTS, and our scope increases, we 'discover' the definition of function $\Delta_{\mathscr{M}}$ (note that $g_{\mathscr{T}}(s) = \Delta_{\mathscr{M}}(\mathscr{I}, \{s\})$). In most of this article, we omit the scope of the cumulated cost function, since it follows from the context. Definition 4 highlights the practice of discovering the final function g (i.e. the total function $g_{\mathscr{T}}$), and, with that, $\Delta_{\mathscr{M}}$, on-the-fly.

Next, in Definition 5, we define *monotonicity* of cumulated cost functions.

Definition 5 (*Monotonicity of cumulated cost functions*). A cumulated cost function $g_T : \mathscr{S} \rightarrow \mathbb{K}$ with scope $T \subseteq \mathscr{T}$ is called *monotonic* iff for all $s \in \mathscr{S} \setminus \mathscr{I}$ there exist $s' \in \mathscr{S}, \ell \in \mathscr{A}$ with $s' \xrightarrow{\ell} s \in T$ and $g(s) \geq g(s')$.

A function g_T is called monotonic if for every state s reachable in T from $\mathscr{I}$, there exists a predecessor with a cumulated cost smaller than or equal to $g_T(s)$. We cannot claim that all predecessors of s have a smaller cumulated cost, because $g(s)$ might very well have been updated to a smaller value at least once during the search, while the g-values of some of its predecessors are not. An example of this is illustrated in Fig. 1. At first, on the left of the figure, the cumulated cost of state s_3 is greater than the cumulated cost of state s_1. However, the transition from state s_2 to s_3 has not been explored yet. When increasing the scope to $T \cup \{s_2 \longrightarrow s_3\}$, we revisit s_3 by expanding s_2, and find that its cumulated cost should be updated. As the cumulated cost of s_1 is not updated, since it is not reachable from s_3, we have $g_{T\cup\{s_2\longrightarrow s_3\}}(s_1) > g_{T\cup\{s_2\longrightarrow s_3\}}(s_3)$.

When considering $\mathbb{K}$ without negative elements, monotonicity of g follows trivially from Definition 4.

Best-first Search: now we can define *best-first search* (BEFS) for WLTSs, where a *guiding function f* is used to determine which states to explore first. In general, the definition of f is left completely open in BEFS, hence BEFS constitutes a type of search, not a concrete individual search. In BEFS for WLTSs, the cumulated cost function is often used in f. In order to optimise the computational complexity, we store $g(s)$ with each state $s \in \mathscr{S}$. We refer to such a stored g-value as $s.g$ (for instance see the pair $\langle s, s.g\rangle$ in Algorithm 1). By doing so, we can compute $g(s)$ for any $s \in \mathscr{S}$ by accessing the stored g-value of (one of) its predecessor(s), and adding the cost of the fired action to it. For this purpose, we redefine $nxt_{\mathscr{M}}(s, T)$ as $nxt_{\mathscr{M}}(s, T) = \{\langle s', s.g + c\rangle \mid s' \in \mathscr{S} \wedge \exists \ell \in \mathscr{A}.\ (s \xrightarrow{\ell} s' \in T \wedge \mathscr{C}(\ell) = c)\}$. The result is presented in Algorithm 1. Here, we leave open how the functions f and g relate to each other. Later on, we will consider possible relationships between f and g. In the algorithm, a selection of β states based on f is done with the function $select_{f,\beta} : 2^{\mathscr{S}} \rightarrow 2^{\mathscr{S}}$, which employs a given *selection width* $\beta \in \mathbb{N} \cup \infty$; let S be a set of states, then $select_{f,\beta}(S) \subseteq S$ and $|select_{f,\beta}(S)| \leq \beta$. Furthermore, each level i of the search is represented by $\mathscr{L}_i \subseteq \mathscr{S}$. In each round of the algorithm, f is used to select a set of states $\mathscr{L}_i \subseteq \mathscr{H}$ from the search horizon $\mathscr{H}$, which consists of all states which have already been visited, but not yet explored. Notice that after generation of the successor states, which are placed in the set $\mathscr{L}_{i+1}$, some states are removed again, namely those which have been encountered before with a lower or equally valued cumulated cost. These states are present in the union of the previous levels, i.e. $\bigcup_{j=0}^{i-1} \mathscr{L}_j$. If, however, in an earlier encounter the cumulated cost was greater, then the state should be re-opened. Of course, such a re-opening introduces redundancy in the search, but can often not be avoided if we wish to preserve *cost-optimality* (which is defined later on). This check is referred to as *duplicate detection*. It is an important step in state space search, since it guarantees termination when generating finite state spaces containing cycles. In DMC, this particular step is changed at times to avoid unnecessary exploration in smarter ways, e.g. by using branch-and-bound

Algorithm 1 Best-first search for weighted state spaces

Require: $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$, guiding function f, selection width β, goal states $\mathscr{G}$
Ensure: If found, a trace to a goal state is returned
  $i \leftarrow 0$
  $\mathscr{H} \leftarrow \{\langle s, 0\rangle | s \in \mathscr{I}\}$
  while $\mathscr{H} \neq \emptyset$ **do**
    $\mathscr{L}_i \leftarrow select_{f,\beta}(\mathscr{H})$
    if $\{s | \langle s, s.g\rangle \in \mathscr{L}_i\} \cap \mathscr{G} \neq \emptyset$ **then**
      return $generateTrace(\{s | \langle s, s.g\rangle \in \mathscr{L}_i\} \cap \mathscr{G})$
    for all $\langle s, s.g\rangle \in \mathscr{L}_i$ **do**
      $\mathscr{H} \leftarrow \mathscr{H} \cup nxt_{\mathscr{M}}(s, en_{\mathscr{M}}(s))$
    $i \leftarrow i + 1$
    $\mathscr{H} \leftarrow \{\langle s, s.g\rangle \in \mathscr{H} | \neg\exists g' \leq s.g. \langle s, g'\rangle \in \bigcup_{j=0}^{i-1} \mathscr{L}_j \wedge \neg\exists g' < s.g.\ \langle s, g'\rangle \in \mathscr{H}\}$
  return *false*

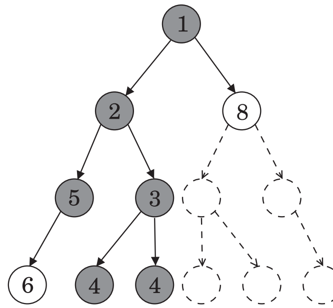

Fig. 2. Example Ucs.

(BnB) [38]. Note that duplicate detection considers multiple appearances of a state in $\mathscr{H}$ itself. When this occurs, only the smallest g-value should be maintained. Whenever a goal state is encountered, we stop the search by invoking the function *generateTrace*, which returns a trace leading from $\mathscr{I}$ to this goal state. If such a goal state does not exist, *false* is returned. [1]
Concerning the usage of $select_{f,\beta}$, we can first of all distinguish two cases with $\beta = 1$ and $\beta > 1$, the first leading to explicit-state searches, and the second to set-based searches. Furthermore, for a given set of states S, consider the subset of states with minimal f-value $S_{min} = \{s \in \mathscr{H} | \forall s' \in \mathscr{H}. f(s) \leq f(s')\}$. Whenever $\beta > |S_{min}|$, we can distinguish two types of $select_{f,\beta}$ functions: either $select_{f,\beta}$ selects exactly S_{min}, i.e. it is limited to selecting states from S_{min}, or additional states are selected from $S \setminus S_{min}$ based on f until β states are selected (unless $|S| < \beta$, in which case S is selected entirely). The first type of $select_{f,\beta}$ functions is used in, e.g. set-based A^*, where in each round, exactly S_{min} is selected, while the second type is used in Bs.

Note that duplicate detection, in the last line of the algorithm, is now refined to the removal of states from the search horizon $\mathscr{H}$ which have been encountered before with a lower or equally valued cumulated cost.

Uniform-cost Search: Now we come to a practical instance of Befs. Given a monotonic cumulated cost function $g : \mathscr{S} \rightarrow \mathbb{K}$, if we say that $f = g$, then Algorithm 1 denotes what is referred to by, e.g. [36,48,56] as *uniform-cost search* (Ucs), and as *lowest-cost-first search* by [54]; it is also known as *Dijkstra's search*, technically if $\mathscr{G} = \emptyset$, from [16]. In Fig. 2, an example of Ucs through a tree is shown, where the g-values are displayed inside the nodes. The grey nodes are selected for exploration, while the obscured ones are nodes that would have been encountered, had their parents been selected. In Ucs, nodes are expanded in order of their g-values. Note that in Algorithm 1, the duplicate detection is unnecessarily complicated for Ucs. This follows from the following observation concerning Ucs:

Lemma 1. *If we consider monotonic g for* Ucs*, then for any states $s, s' \in \mathscr{S}$, if s is selected by the $select_{g,\beta}$ function in round i, and s' is selected by the $select_{g,\beta}$ function in round $j > i$, then necessarily $s.g \leq s'.g$. This is true independent of the value of β.*

Since Lemma 1 also holds when $s = s'$, it follows that in duplicate detection it suffices to check for earlier encounters of a state, independent of its g-value. More formally, we can simplify duplicate detection to $\mathscr{H} \leftarrow \{\langle s, s.g\rangle \in \mathscr{H} | \neg\exists g'. \langle s, g'\rangle \in \bigcup_{j=0}^{i-1} \mathscr{L}_j\}$.

[1] The decision to have Algorithm 1 and subsequent algorithms return *false* in case no goal state is encountered was made interpreting goal states as desirable states.

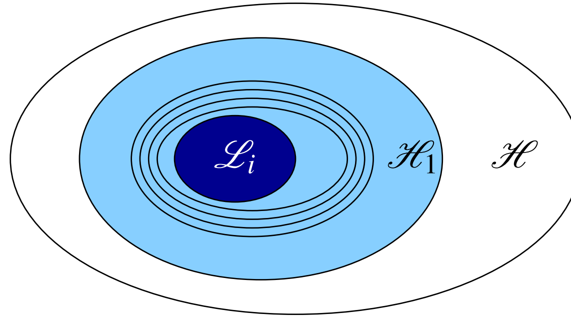

Fig. 3. Multi-phase selection in M-BEFS.

From Lemma 1 it follows that the first time a state $s \in \mathscr{G}$ is discovered, a trace from $\mathscr{I}$ to s is discovered with $g(s) = \Delta_{\mathscr{M}}(\mathscr{I}, \{s\})$. Actually, if it is the very first goal state we encounter, then $g(s) = \Delta_{\mathscr{M}}(\mathscr{I}, \mathscr{G})$.

We call the second property *cost-optimality*, which is defined in Definition 6.

Definition 6 (*Cost-optimality of a best-first search*)**.** Given a WLTS $\mathscr{M} = (\mathscr{S}, \mathscr{A}, \mathscr{C}, \mathscr{T}, \mathscr{I})$ and a set of goal states $\mathscr{G} \subseteq \mathscr{S}$, we call a BEFS *cost-optimal* iff it is ensured that it always returns a trace from $\mathscr{I}$ to a goal state $s \in \mathscr{G}$ with $f(s) = \Delta_{\mathscr{M}}(\mathscr{I}, \mathscr{G})$ (unless $\mathscr{G} = \emptyset$ or unreachable).

For instance, in general, BFS, which is called a *blind* or *uninformed* search since it ignores g, applied on a WLTS (which is represented in Algorithm 1 by choosing an appropriate f, such as $f(s) = 0$ **if** $s \in \mathscr{I}$, and $f(s) = f(s') + 1$, **if** $\exists \ell \in \mathscr{A}. s' \xrightarrow{\ell} s$,[2] and furthermore having $\beta = \infty$), is not cost-optimal, unless searching is continued once a trace to a goal state is found until there are no more states to explore. This is because there is no necessary correlation between the length of a trace and its total weight; so the first time a goal state is reached does not necessarily mean that a minimal trace to a goal state is found. On the other hand, in a WLTS where for all $\ell \in \mathscr{A}, \mathscr{C}(\ell) = 1$, BFS is cost-optimal.

Finally, depth-first search (DFS) can be achieved in Algorithm 1 by, e.g. determining f on-the-fly as $f(s) = 0$, **if** $s \in \mathscr{I}$, and $f(s) = f(s') - 1$, **if** $\exists \ell \in \mathscr{A}. s' \xrightarrow{\ell} s$ and setting $\beta = 1$.

3. Multi-phase best-first search

In this section, we propose an extension of BEFS, which we call *multi-phase* BEFS (M-BEFS), which gives rise to the idea of compositionality of BEFSs. This extension is by no means common; in the literature we find some searches which may be identified as instances of M-BEFS, but the general form is not described. The three possible instances we found are *filtered* BS (reported, e.g. by [59,50,64]), A^*_ϵ [48], and *heuristic depth-first search* (e.g. see [54]). Furthermore, in the context of AND/OR graphs, the *General Best-First Strategy* (GBF) is a two-phase BEFS [48]. Besides the existence of these searches, our proposed extension will be furthermore motivated in the sections on BS in this article.

In BEFS, one can manipulate the way in which the $\mathscr{L}_i$ are constructed by means of providing a selection function, a guiding function f and a selection width β. A generalisation would be to allow a list of n selection functions $select^1_{f_1,\beta_1}, \ldots, select^n_{f_n,\beta_n}$, each with their own guiding function f_i and selection width β_i.[3] The list could be used to construct an $\mathscr{L}_i$ through a number of intermediate phases, such that in practice, we basically stack search algorithms on top of each other. We will further illustrate this concept next.

Say we have an n-phase BEFS, with selection functions $select^1_{f_1,\beta_1}, \ldots, select^n_{f_n,\beta_n}$. In each round i of the algorithm, we construct $\mathscr{L}_i$ from $\mathscr{H}$ as follows, which is illustrated in Fig. 3:

1. In the first phase, we use function $select^1_{f_1,\beta_1}$ to select up to β_1 states from $\mathscr{H}$. The selected states together form the intermediate search horizon $\mathscr{H}_1$. If a goal state is found, a trace from $\mathscr{I}$ to this goal state will be produced.
2. In the second phase, we take $\mathscr{H}_1$ and produce the intermediate search horizon $\mathscr{H}_2$ by applying $select^2_{f_2,\beta_2}$ to select up to β_2 states from $\mathscr{H}_1$.
3. ...

n. In the *n*th phase, we take the intermediate search horizon $\mathscr{H}_{n-1}$, and produce $\mathscr{L}_i$ by applying function $select^n_{f_n,\beta_n}$ to select up to β_n states from $\mathscr{H}_{n-1}$.

---

[2] Note that as with cumulated cost functions, f is technically not a function here, as a state s may receive different f-values during a search. It is, however, a (partial) function at any specific moment of the search.

[3] In general, there are other possible parameters for BEFS, not explicitly mentioned in this article, such as a cost upper bound U, which is used to prune away states s with $g(s) > U$. In a similar fashion, a list of cost upper bounds can be provided for M-BEFS.

States selected in phase $j < n$ are not directly removed from $\mathscr{H}_0$, since these may very well not be selected in subsequent phases, in which case we should reconsider them in the next level of the search. The final selection of the level, though, once phase n has finished, is removed from the search horizon, since these states are going to be fully explored.

By adopting such a multi-phased mechanism, we can compose search algorithms useful for several reasons. For instance, the aforementioned filtered Bs can be seen as a two-phase BeFs, where in the first phase states are selected based on some computationally cheap guiding function, which does not incorporate the history of the states, while the remaining states are pruned away. In the second phase, a second selection is performed, and pruning is again applied, using a more precise, but computationally much more expensive, guiding function. By this approach, we can avoid evaluating all states in the original search horizon with the computationally more expensive guiding function.

One can also imagine combining cost-optimal searches. For instance, an n-phase Ucs could be useful to deal with so-called *multi-cost problems* (e.g. [6,40]). As an example, consider the problem of constructing a new building, where money, time and manpower are the three types of resources to consider. The goal is to find a way to construct the building, such that the *amount of money* needed should absolutely be minimised. Given this condition, the *quickest* possible solution should be chosen, and finally, given those two conditions, we should try to minimise the *amount of manpower* needed. Such a solution, in the WLts of the specification of this problem represented by a trace, could be found by using a three-phase Ucs, where $f_1(s) = g_1(s)$ keeps track of the amount of money spent, $f_2(s) = g_2(s)$ reports the time needed thus far, and $f_3(s) = g_3(s)$ reflects the total amount of manpower. One can imagine that changing the priorities of these three types of resources leads to different kinds of solutions, and that changing the order of the phases in the multi-phase Ucs allows us to deal with these different priorities. Note that such multi-cost problems are different from so-called *pareto-optimal* problems, as dealt with in, e.g. [23]. There, the optimal trade-off needs to be found, i.e. a solution such that there is no other solution which is more cost-effective according to some requirement, while not being worse according to the others.

On a side note, it should be pointed out that multi-phase searches raise the interesting question how duplicate detection should be performed, or more specifically, when to re-open states and when not. For instance, if we open a state s, after computing both $f_1(s)$ and $f_2(s)$, and later, we re-encounter s, this time with a lower $f_1(s)$ value, but with a greater $f_2(s)$ value, what to do next? Of course, the re-opening policy should be decided based on the importance of the individual guiding functions.

Returning to the previously mentioned instances of M-BeFs, as explained before, filtered Bs applies two different types of Bs in two phases, subsequently, where in the first phase a computationally cheap guiding function is used, and the second phase deals with a more thorough selection among the remaining states. This search is described in more detail in Section 5.5. The A^*_ϵ search can be seen as a multi-phase search, where in the first phase, standard (set-based) A^* with selection of extra states is used to make an intermediate selection of states. In this phase, all states need to be selected with an f-value not greater than the minimal f-value plus a pre-given value ϵ. In the second phase, $f_2(s) = h_2(s)$, where h_2 expresses a *search effort* estimate. The search effort estimate must not be confused with the estimated remaining cost along a trace; it concerns the remaining computational effort needed by the search algorithm to find a goal state. This second estimate is used to make a final selection of states for exploration from the intermediate set of states. This approach gives rise to the notion of *ϵ-admissibility*; an algorithm is ϵ-admissible iff it is guaranteed to find a solution not worse than the optimal solution plus ϵ. This notion is derived from the notion of admissibility of algorithms, which states that an algorithm is called admissible iff it is guaranteed to find an optimal solution, i.e. it is cost-optimal.

Finally, heuristic DFs is a DFs that uses an estimation function to decide for each s the order in which the successors need to be explored. Note that the estimation function is used locally per state here. A global way, considering the whole search horizon, would constitute a search which could be called heuristic BFs. Heuristic DFs can be seen as a two-phase BeFs, where the first phase selects the states as a DFs would, yielding the set of successors S of one state. For this, we can determine f_1 on-the-fly as $f_1(s) = 0$, **if** $s \in \mathscr{I}$, and $f_1(s) = f_1(s') - 1$, **if** $\exists \ell \in \mathscr{A}.s' \xrightarrow{\ell} s$. In the second phase, an estimation function $h : \mathscr{S} \to \mathbb{K}$ is applied as f_2 to select one state from S. The concept of M-BeFs will reappear in subsequent sections when dealing with so-called *G-synchronised* Bs for WLtss.

4. Beam search

Bs (e.g. [59,7,50]) is a heuristic search algorithm for combinatorial optimisation problems, which was originally used in the artificial intelligence community by Lowerre [42] for speech recognition, and by Rubin [55] for image understanding. Later on, this technique has been applied to scheduling problems, e.g. in [26,61,58] in systems designed for jobshop environments (for an explanation of this kind of problem in a model checking context, e.g. see [67,69]). Since then, new variants of Bs, such as filtered Bs [59,60,50] and recovery Bs [15,64], have been introduced.

Bs is similar to BFs as it progresses level by level. At each level of the search tree, it uses a heuristic evaluation function to estimate the promise of encountered nodes,[4] while the goal is to find a path from the initial state (initial node of the tree) to a leaf node that possesses the minimal evaluation value among all the leaves. At each level, only the β most promising nodes are selected for further examination and the other nodes are permanently discarded. The *beam width* parameter β is fixed

[4] In this section, we use the common Bs terminology, i.e. we reason about nodes and edges, as opposed to states and transitions. This emphasises that we adapt the Bs techniques to a different setting.

1. Set B = ∅, C = ∅
   - Branch n_0 to generate its children
   - Perform priority evaluation of each child node
   - Select min$\{\beta$,number of children$\}$ best child nodes, add them to B
2. For each node in B:
   - Branch node to generate its children
   - Perform priority evaluation of each child node
   - Select best child node, add it to C
3. Set B = C; Set C = ∅
4. Stopping Condition: if all nodes in B are leaf, select node with lowest total-cost and stop, otherwise go to step 2.

Fig. 4. PBS for search trees [63].

1. Set C = ∅, B = $\{n_0\}$
2. For each node in B:
   - Branch node to generate its children
   - Perform detailed evaluation of each child node n (i.e. calculate $f(n) = g(n) + h(n)$)
   - Select min$\{\beta$,number of children$\}$ best child nodes, add them to C
3. Set B = ∅; Select min$\{\beta, |\mathrm{C}|\}$ best nodes in C, add them to B; Set C = ∅
4. Stopping Condition: if all nodes in B are leaf, select node with lowest total-cost and stop, otherwise go to step 2.

Fig. 5. DBS for search trees [63].

to a value before searching starts. Because of this aggressive pruning, the generation time is linear to the maximum search depth, and is thus heavily decreased. However, since wrong decisions can be made while pruning, BS is neither complete nor cost-optimal. To limit the possibility of wrong decisions, one can increase the beam width, at the cost of increasing the required computational effort and memory use.

The original definition of BS allows any kind of guiding function to be used. Using the terminology of [67], it only demands that pruning in the width is performed, and extra states are selected (see Section 5.2); as [26] put it: "[BS] builds a highly pruned search tree of labelling alternatives which resembles a beam". In this article, however, we focus on two types of evaluation functions, which have traditionally been used often for BS, as, for instance, reported in [59,64]: *priority* evaluation functions and *total-cost* evaluation functions, which lead to the *priority* (PBS) and *detailed* (DBS) BS variants, respectively. In PBS, at each node a priority evaluation function calculates a priority for each successor node, and the algorithm selects based on those priorities. At the root of the search tree, up to β most promising successors (i.e. those with the highest priorities) are selected, while in each subsequent level only one successor with the highest priority is selected per examined node. Fig. 4 describes the basic idea of traditional PBS. There, n_0 is the root of the search tree, and all leaves are assumed to be located at the same level.

In DBS, at each node n, the evaluation function $f(n) = g(n) + h(n)$ calculates an estimate of the minimal weighted distance between n_0 and the set of goal nodes $\mathscr{G}$ via n. For this, the cumulated cost of n is added to an estimate of the minimal weighted distance between n and $\mathscr{G}$. At each level, up to β most promising nodes (i.e. those with the lowest total-cost values) are selected, regardless of who their parent nodes are. If there are more than β nodes that receive the best evaluation value, a selection is made based on other criteria, e.g. the order of encountering the nodes (see Section 5.4 for other possibilities). Clearly, when $\beta \to \infty$, DBS and PBS behave as exhaustive BFS. Fig. 5 represents traditional DBS.

In comparison, priority evaluation functions have a local view of the problem, since they only consider the next node to be selected, while total-cost evaluation functions have a more global view, taking complete traces from n_0 to $\mathscr{G}$ into account and comparing different branches of the tree. The intuition behind priority evaluation functions is that one cannot simply compare priorities of nodes which do not have the same parent node, because the suitability of a node depends on what came before in the trace from n_0 to the parent. This is closely linked to the fact that PBS is typically used on trees representing jobshop scheduling problems, where a finite number of jobs $\{t_1, \ldots, t_n\}$ $(n \geq 1)$ need to be performed in some order. Then, each node represents performing a job, but which jobs can be performed next depends on the jobs that have previously been performed. Say we have a priority function $prio : \mathscr{N} \to \mathbb{Z}$ to guide the search, with $\mathscr{N}$ the set of nodes. In a tree representing a jobshop problem, every trace leads to a goal node $n \in \mathscr{G}$, and along a trace, each job is performed exactly once. Within such a tree, we wish to guide the search by associating priorities with the job executions, i.e. the nodes, thereby stimulating the early execution of highly important jobs. Now, let us look at nodes $n_0, \ldots, n_6$ in Fig. 6a.

Say that n_2 and n_4 are associated with performing job t_i, and n_3 and n_6 with performing job t_j, for $(0 \leq i, j \leq n)$ and $(i \neq j)$. Clearly, from node n_1, where both t_i and t_j can be performed next, if we need to select one node based on *prio*, and $prio(t_i) > prio(t_j)$, we will choose t_i. If we would take all the successor nodes of both n_1 and n_5 together into account, since

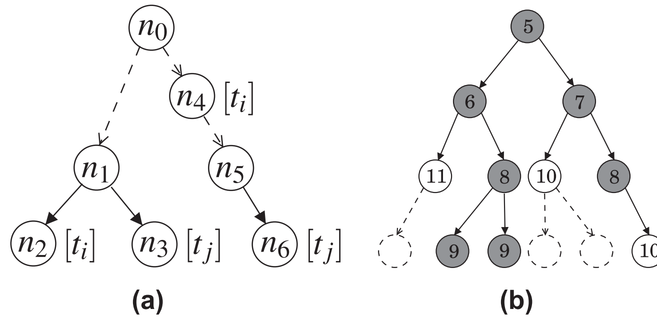

Fig. 6. (a) Selection of transitions. (b) Example DBS with $\beta = 2$ in a search tree.

these are on the same search level, and make a (global) selection based on *prio*, we would make a mistake. Note that in the trace leading from n_0 to n_5, t_i has already been performed once in n_4. In other words, in n_5 we are in a completely different situation; in fact, it might be that n_5 represents a situation in which we have made more progress with the problem than in n_1, thereby we only have jobs left with lower priorities. As we do not want to delay or even prune away this more promising trace, histories of nodes are important when selecting, and therefore we may only compare nodes with each other if they have the same parent.

The inability of priority functions to globally compare nodes may be a limitation of this kind of functions, compared to total-cost functions, but priority functions also have advantages; first of all, their computational complexity is, in general, far less. Priority functions usually only encompass looking up a priority for a given job (e.g. in a table), while cost functions may incorporate complex calculations, particularly when heuristics are involved. Second of all, focussing on our desire to apply BS on LTSs, no cost values are involved with a priority function, therefore, such a function could be applied on an unweighted LTS, which implies that a cheaper duplicate detection suffices, in which g-values are not checked. However, total-cost evaluation functions often provide more accurate heuristics because of their global view [59].

Fig. 6b shows the application of a DBS on a search tree, where the f-values are displayed in the nodes. Typically, this is a DBS as opposed to a PBS. First of all, in a DBS, states with the lowest f-values are selected, while in a PBS, priorities must be high in order to qualify for selection. Second, in a PBS, up to β transitions from the root of the tree are followed, after which in each subsequent level of the tree one outgoing transition with the highest priority is selected per examined node. In a DBS, however, at each level up to β nodes are selected to continue, regardless of what their parent nodes are, therefore it could be the case, as in level 4 of Fig. 6b, that some nodes have multiple selected children, while others have none.

5. Adapting beam search for state space generation

5.1. Motivation

BS is typically applied on highly structured search trees, e.g. in [47,64]. If we encode a jobshop problem, as explained in Section 4, in a WLTS, in which the transitions have labels, it is quite intuitive to label the transitions with the executed jobs. The WLTS in Fig. 7a contains all possible orderings of jobs $\{t_1, t_2, t_3\}$. Such a search tree starts with n jobs to be scheduled, which means that the root of the tree has n outgoing transitions. Every node has exactly $n - k$ outgoing transitions, where k is the level in the tree where the node appears. However, in general, WLTSs contain information on all possible behaviours of a system, and this may incorporate cycles or confluence of traces (i.e. states having multiple incoming transitions), i.e. WLTSs can have more complex structures than the well-structured search trees usually subjected to BS. This necessitates modifying the BS techniques to deal with arbitrary structures. Moreover, the BS algorithms search for a particular node in the search tree, while in (and after) generating WLTSs one might desire to study a property beyond simple reachability (see Section 10.2 on POR as an instance of extended BS). We therefore extend BS to a state space *generation* setting, as opposed to its traditional setting that focuses only on *searching*. See Section 7 for possible optimisations when restricting BS to verify reachability properties. This, along with the necessary machinery for handling cycles, raises memory management issues in BS, as we will see in Section 7.

First, we adapt PBS and DBS for state space generation. Next, we propose two variants of BS which have, in our case studies, proved essential for handling large WLTSs. Flexible BS mitigates the problem of determining a sufficiently large beam width, while synchronised BS separates the pruning phase from the exploration strategy.

Fig. 7b shows the spectrum of the variants that are described in the following sections. Sections 5.2 and 5.3 deal with PBS and DBS, respectively. The F and S prefixes refer to the flexible and synchronised BS variants (Sections 5.4 and 5.5).

5.2. Priority beam search for state space generation

Next, we motivate and describe the changes that we have made to the traditional PBS to deal with state space generation.

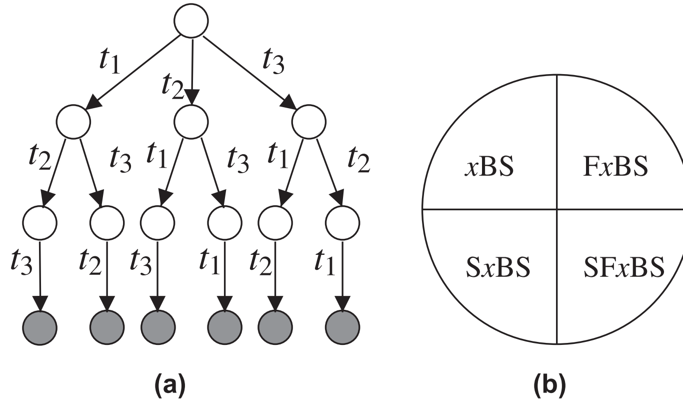

Fig. 7. (a) Search tree for a single-resource scheduling problem with tasks t_1, t_2, t_3. (b) Bs spectrum, $x \in \{D, P\}$.

Algorithm 2 PBS for WLTSS

Require: $\mathscr{M}=(\mathscr{S},\mathscr{A},\mathscr{T},\mathscr{I})$, widening factor α, stabilisation level l, priority function $prio : \mathscr{A} \rightarrow \mathbb{Z}$, set of goal states $\mathscr{G}$
Ensure: If found, a trace to a goal state is returned

```
i ← 0
𝓛_i ← 𝓘
Buffer ← ∅
limit := α
while 𝓛_i ≠ ∅ do
  𝓛_{i+1} ← ∅
  if 𝓛_i ∩ 𝓖 ≠ ∅ then
    return generateTrace(𝓛_i ∩ 𝓖)
  for all s ∈ 𝓛_i do
    for all s −ℓ→ s′ ∈ en_𝓜(s) do
      if prio(ℓ) > prio_min(Buffer) then
        if |Buffer| = limit then
          Buffer ← Buffer \ {getprio_min(Buffer)}
        Buffer ← Buffer ∪ {s −ℓ→ s′}
    𝓛_{i+1} ← 𝓛_{i+1} ∪ nxt_𝓜(s, Buffer)
    Buffer ← ∅
  𝓛_{i+1} ← 𝓛_{i+1} \ ⋃_{j=0}^{i} 𝓛_j
  i ← i + 1
  if i = l then
    limit := 1
return false
```

PBS is shown in Algorithm 2. The user-supplied function $prio : \mathscr{A} \rightarrow \mathbb{Z}$ provides the priority of actions, as opposed to states. We motivate this deviation from the traditional notion of PBS by noting that *jobs* in the scheduling terminology correspond more naturally with *actions* in a WLTS when specified for a model checker, as already mentioned earlier. The finite set of actions $\mathscr{A}$ then nicely corresponds with the set of jobs to perform. Moreover, by applying the priority function on actions, we can use such functions for any WLTS, instead of only the ones representing a jobshop problem, since all WLTSS have labelled transitions, and there is therefore no need for an additional function mapping states with the prioritised elements (e.g. jobs).

The set *Buffer* temporarily keeps seemingly promising transitions. The function $prio_{min} : 2^{\mathscr{T}} \rightarrow \mathbb{Z} \cup \{-\infty\}$ returns the lowest priority of the actions of a given set of transitions. We define $prio_{min}(\emptyset) = -\infty$. The function $getprio_{min} : 2^{\mathscr{T}} \setminus \emptyset \rightarrow \mathscr{T}$, given a set T of transitions, returns one of the transitions in T labelled by an action with priority $prio_{min}(T)$. The aforementioned functions are used together in Algorithm 2 to select a subset of $\mathscr{L}_i$, but in two important ways this differs from the $select_\beta$ function of, e.g. UCS:

1. Not only the very best states are selected according to the evaluation function. Extra states may be selected (considered in order of evaluation value), as long as the selection limit has not been reached.
2. All the states not selected are pruned from memory; they are not reconsidered in later rounds (unless visited again).

In the terminology of [67], we call these algorithmic properties *selection of extra states* and *pruning in the width*, respectively.

Note that the stopping condition of the traditional PBS algorithm of Section 4 is represented here by the condition $\mathcal{L}_i \neq \emptyset$ of the **while** loop, which does not assume that all leaves occur in the same level (this, moreover, avoids cycles). The algorithm terminates when it has explored all the states in its beam.

In PBS, originally, up to β children of the root are selected. The resulting beam of width β is then maintained by sprouting only one child per node in subsequent levels. This works for trees such as the one shown in Fig. 7a, where the root has more outgoing transitions than any other node in the tree. In WLTSs, however, the root has typically considerably fewer outgoing transitions than the average branching factor of the WLTS. Fixing the beam width at such an early stage is therefore not reasonable.

Selecting all transitions at each level until β or more transitions are found in a single level would be an option. However, if this number drastically exceeds β, it would not be clear which transitions should be pruned away. To mitigate this problem, instead of β, Algorithm 2 is provided with the pair (α, l), where $\alpha, l \in \mathbb{N}$ and $\alpha^l = \beta$. We call α the *widening factor* and l the *stabilisation level*. The idea is that the algorithm uses the *prio* function to prune non-promising states from the very first level, but in two phases: before reaching β states in a single level it considers per state the most promising α transitions for further expansion, but after that (i.e. once it has reached level l), it sticks to the one successor per state rule. Here, the assumption is made that in the first l levels of the WLTS, each state has at least α outgoing transitions. In practice, this is not such a strong assumption, considering that the kind of problems for which BS is suitable typically produces WLTSs which resemble trees that expand quickly.

5.3. Detailed beam search for state space generation

The original idea of DBS does not need to change much to fit into the state space generation setting, except for handling cycles; we need to incorporate the weight-sensitive duplicate detection of Algorithm 1. As a side note, the average running time of the DBS algorithm of Section 4 can be reduced if the order of exploration and evaluation is reversed. Intuitively, instead of first expanding the nodes of the current level and then evaluating the children and selecting the β most promising of them to constitute the next level (cf. the algorithm of Section 4), we first evaluate the states of the current level, select the β most promising states among them and then expand them, to constitute the next level. When performed successively, these two orders are identical. However, since the number of nodes to be evaluated is a priori known in each level, evaluation of the states of a level containing no more than β states can altogether be avoided.[5]

Besides that, to reduce the space complexity of the traditional DBS algorithm of Section 4, while evaluating, only the β most promising states can be kept in a set and the rest can be discarded (note that β^2 states are stored in the algorithm of Section 4). This optimisation, of course, does not depend on the order of evaluation and exploration.

DBS is represented by Algorithm 1 if we define $select_{f,\beta}$ as in Algorithm 3. Function f is decomposed into $f(s) = g(s)+h(s)$, in other words, as explained in Section 4, it incorporates a cumulated cost part and a heuristic part. The $g(s)$ function represents the cumulated cost taken to reach s from $\mathcal{I}$, as defined in Definition 4. The cost-function $\mathcal{C}$ is user-supplied. The weights can, e.g. denote the time needed to perform different jobs in a scheduling problem. These weights are fixed before searching starts.

The user-supplied heuristic function $h(s)$ estimates the cost it would take to efficiently complete the schedule continuing from s. Similar to g, the total-cost function f is called *monotonic* iff $s \rightarrow^* s' \implies f(s) \leq f(s')$.

The function $getf_{min} : 2^{\mathcal{S}} \setminus \emptyset \rightarrow \mathcal{S}$, given a set of states, returns one of the states that has the smallest f-value. It thus computes $f(s) = g(s) + h(s)$ for each member of the set (it can of course be optimised for consecutive calls to the same set). Note that in Algorithm 3, pruning is applied, as all states not selected are removed from the horizon.

Algorithm 3 $select_{f,\beta}$ for DBS for WLTSs

Require: search horizon $\mathcal{H}$, guiding function $f : \mathcal{S} \rightarrow \mathbb{K}$, beam width β
Ensure: A subset $\mathcal{L} \subseteq \mathcal{H}$ is returned with smallest g-values, $|\mathcal{L}| \leq \beta$
  $\mathcal{L} \leftarrow \emptyset$
  while $|\mathcal{L}| < \beta \wedge \mathcal{H} \neq \emptyset$ **do**
    $\mathcal{L} \leftarrow \mathcal{L} \cup \{getf_{min}(\mathcal{H})\}$; $\mathcal{H} \leftarrow \mathcal{H} \setminus \{getf_{min}(\mathcal{H})\}$
  $\mathcal{H} \leftarrow \emptyset$
  return $\mathcal{L}$

5.4. Flexible beam search

A major issue that still remains unaddressed in the BS adaptations of Sections 5.2 and 5.3 is how among equally competent candidates, i.e. having the same f-values, pruning should be carried out.

[5] Actually, the evaluation of the heuristic estimation part, which is computationally the most expensive phase, is the part that is avoided. See Algorithm 3 for details.

Actions in WLTSs can have several parameters. The same action can thus appear multiple times as an outgoing transition of a given state, each time having different parameter values, possibly leading to equally competent states. This potentially leads to situations where, during selection, a large number of transitions or states have equal evaluations (for some examples, see Section 9). In such cases, a selection has to be made among equally competent candidates if they happen to be (one of) the most promising transitions or among the β-best states. These selections are beyond the influence of the evaluation (or priority) function and can undesirably make the algorithm non-deterministic. Hence, we propose two variants of Bs that we call *flexible detailed* and *flexible priority* Bs, in which the beam width can change during state space generation.

In flexible DBS, at each level, up to β most promising states are selected, plus any other state which is as competent as the worst member of these β states. This achieves closure on the worst (i.e. highest) total-cost value being selected. Similarly, in flexible PBS, in the first l levels (see Section 5.2), at each state, up to α most promising outgoing transitions are selected, plus any transition which has the same priority as the least competent member of these α transitions. At the $l + 1$th level and onwards, at each state, all the transitions with the same priority as the most promising transition of that particular state are selected (i.e. as if $\alpha = 1$). In other words, in flexible BSs, tie-breaking is avoided, by making the beam dynamic in size. Note that in flexible PBS, in contrast to flexible DBS, if the beam width is stretched, it cannot be readjusted to the intended β.

The benefit of this approach is that there are no selection criteria other than the evaluation function used. This not only leads to more insight in the effectiveness of the function, but in practice it may also mean that smaller beam widths can be used, compared to non-flexible Bs (see, for instance, the results in Section 9). The drawback is that the memory requirement is no longer linear in the maximum search depth, since β is only a guideline for the beam width.

5.5. G-synchronised beam search

As is described in Section 4, the classic Bs algorithms were tailored for the BFS strategy. Next, we explain a more general setting in which Bs can be used with any best-first exploration strategy. Broadly speaking, we separate the exploration strategy from the pruning phase, where the exploration may be guided with a different evaluation function. This is particularly useful when checking reachability properties on-the-fly.

We inductively describe *G-synchronised x*-Bs as a two-phase BEFS, where G is the function that guides the exploration and x can be either *detailed*, *priority*, *flexible detailed* or *flexible priority*. Let $\mathscr{H}$ be the current search horizon. In the exploration phase of round i, we need to determine the set of states to be explored $\mathscr{L}_i \subseteq \mathscr{H}$. We do this by first determining an intermediate search horizon $\mathscr{H}'$ by employing the guiding function G as follows: $\mathscr{H}' = \{s \in \mathscr{H} | \forall s' \in \mathscr{H}. G(s) \leq G(s')\}$. Subsequently, the pruning phase of x-Bs is applied on $\mathscr{H}'$, leading to $\mathscr{L}_i \subseteq \mathscr{H}'$. According to the pruning phase (which can possibly employ an evaluation function different from G), some of the states in $\mathscr{H}'$ are selected, constituting the set $\mathscr{L}_i$. Finally, the successors of all the states in $\mathscr{L}_i$ are determined and added to $\mathscr{H}$. The next round starts with the search horizon $\mathscr{H} \setminus \mathscr{H}'$ and needs to determine $\mathscr{L}_{i+1}$. Algorithm 1 describes this technique if we define $select_{f,\beta}$ as in Algorithm 4. Since this technique distinguishes an exploration phase and a pruning phase, it can combine most exploration strategies (from different BEFSs) with all the variants of Bs introduced earlier. Using any constant function as G in synchronised DBS clearly results in Bs with a BFS exploration strategy.

Algorithm 4 $select_{f,\beta}$ for G-synchronised DBS for WLTSs

Require: search horizon $\mathscr{H}$, exploration function G, heuristic function $h : \mathscr{S} \to \mathbb{K}$,
beam width β, set of goal states $\mathscr{G}$
Ensure: $\mathscr{L} \subseteq \mathscr{H}' \subseteq \mathscr{H}$ is returned, with $\mathscr{H}'$ the set of elements from $\mathscr{H}$ with smallest G-values, $\mathscr{L}$ the set of elements from $\mathscr{H}'$ with smallest h-values, $|\mathscr{L}| \leq \beta$
$\mathscr{L} \leftarrow \emptyset$; $\mathscr{H}' \leftarrow \{\langle s, s.g\rangle \in \mathscr{H} | \forall \langle s', s'.g\rangle \in \mathscr{H}. G(s) \leq G(s')\}$
while $|\mathscr{L}| < \beta \wedge \mathscr{H}' \neq \emptyset$ **do**
  $\mathscr{L} \leftarrow \mathscr{L} \cup \{getf_{min}(\mathscr{H}')\}$; $\mathscr{H}' \leftarrow \mathscr{H}' \setminus \{getf_{min}(\mathscr{H}')\}$
$\mathscr{H} \leftarrow \mathscr{H} \setminus \mathscr{H}'$
return $\mathscr{L}$

Modular implementation of synchronised Bs variants can thus be conceived: the first phase takes care of the order in which states need to be considered for pruning and exploration, and the second phase performs the actual pruning and selection. Such a two-phase approach resembles filtered Bs, described by [59], where classic PBS is applied before classic DBS takes place. In Section 3, we described a general algorithm, encompassing these two types of searches, called M-BEFS, which can also deal with more than two phases per round.

To mention a practically interesting candidate for G, we temporarily return to the application of finding schedules for a given problem. This means that we wish to find a path of minimal cost that leads to a particular action or state in a WLTS. If for every action ℓ, $prio(\ell) = 1$, this problem corresponds to finding the minimal length trace when verifying a reachability property. Recall that the total-cost function in DBS can be decomposed into $f(s) = g(s) + h(s)$, where $g(s)$ is the cost of the trace leading from $\mathscr{I}$ to s. If $G(s) = g(s)$, then we practically have a DBS pruning mechanism together with a UCS exploration strategy. As it turns out, this means that UCS properties such as Lemma 1 are inherited; in g-synchronised DBS, once a goal state (or a complete schedule) is found, searching can safely terminate. This is because in a goal state s, $f(s) = g(s)$, and since the algorithm always follows paths with minimal g (remember that g is monotonic), state s is reached before another

state s' iff $g(s) \leq g(s')$. Note that here no state is re-opened, because states with minimal g are taken first and thus a state can be reached again only via paths with greater costs (cf. Section 5.3). This in practice removes the necessity to store costs with states [67,69].

Both g-synchronised DBS and g-synchronised PBS have been used in solving timed scheduling problems, the results of which are reported in Section 9, where minimal-time traces to a particular action label are searched for. One can imagine these searches as two-phase BEFSS with a UCS in the first phase, and a greedy search (BEFS with $f(s) = h(s)$) and a PBS in the second phase, respectively. The same pruning algorithm can be used to search for other kinds of traces, such as a shortest trace or a shortest minimal-time trace.

6. Beam search in the μCRL toolset

In the μCRL toolset, we implemented the g-synchronised variants of BS, such that the exploration phase is performed in an (action-based) minimal-cost manner (see minimal-cost search [67,69], which is a specific kind of UCS for WLTSS where costs are represented by special, additional actions), therefore these variants can be applied on unweighted LTSS with an action-based representation of costs. In these variants, there is no additional space needed to store (intermediate) cumulated cost results for states, and the duplicate detection can be done straightforwardly, not considering the cumulated costs. Next, we describe how total-cost and priority evaluation functions are represented in the toolset.

In our implementation of PBS, priority values are assigned to actions, as opposed to nodes in classic BS, and are fed to the state space generator in an input file. To be precise, priority values are assigned to action *labels* and are fixed during the search. Therefore, identical action labels have equal priority levels regardless of their source, destination or parameters (if present). By default, all actions have priority zero.

Related to DBS, the μCRL toolset can perform a g-synchronised DBS with $f(s) = h(s)$.[6] The desired estimation function $h : \mathscr{S} \to \mathbb{K}$ can be specified using constants and variables taken from the parameter list of the specification, combined with the standard arithmetic operations, i.e. addition, subtraction, multiplication and division. If the estimation function has a sophisticated structure, e.g. depends on some pre-calculated information as in the case of [47], it should be encoded in the specification. In μCRL, abstract data types are used to specify data structures and functions. This is very expressive and allows creating many useful functions, possibly incorporating pre-calculated data; see, e.g. [32].

In general, G-synchronised searches can be applied, as long as the G-guiding is action-based, like e.g. minimal-cost search, which can keep track of g-values by recording the number of encountered cost transitions (here labelled *tick*) along the way. Also, flexible versions of the implemented variants are available.

Of course, alternatively, a total-cost function $f(s) = g(s) + h(s)$ can be achieved by keeping track of cumulated costs of states in a special variable *cost* in the specification, as explained e.g. in [67,69].

Case studies on timed scheduling problems using the BS implementations in the μCRL toolset are discussed in Section 9.

7. Memory management

Memory management is a challenging issue in state space generation. Although BS reduces memory use due to cutting away parts of the WLTS, still explored states need to be accessed to guarantee the termination of the exploration in case of cyclic state spaces. Keeping the whole set of visited states in the memory is usually susceptible to early state space explosion. This can be counter-measured by taking into account specific characteristics of the problem at hand and the properties that are to be checked. Below we discuss some possible optimisations when applying BS:

1. When aiming at a reachability property on-the-fly (such as reachability of a goal state, checking invariants and hunting deadlock states), the memory requirements can be lowered by checking the property while exploring. In that case, once a state satisfying the desired property is reached, the search terminates and the witness trace is reported. This, however, cannot be extended to arbitrary properties.
2. If there are no cycles in the WLTS, there is in principle no need to check whether a state has already been visited (in order to guarantee termination). Therefore, only the states from the current level need to be kept and the rest can be removed from memory, i.e. flushed to high latency media such as disks. In this case, however, some states may be revisited due to confluent traces, hence undesirably increasing the search time. Prominent examples of systems with acyclic WLTSS are large classes of scheduling problems, which have been traditional targets of BS, and most security protocols (see Section 10.2). As is demonstrated by [62], a known POR algorithm for security protocols can be seen as an instance of BS.
3. In DBS variants, if each state s has a unique cumulated cost $g(s)$ associated to it, e.g. denoting a notion of progress, and if g is monotonic, then there cannot be any transition from states with a greater cumulated cost to the states with lower cumulated costs: $g(s) < g(s') \implies s \not\to^* s'$. Consequently, states with cumulated costs strictly lower

[6] In fact, note that a guiding function $f(s) = g(s) + h(s)$ has the same effect here, as in each round of the search, g-synchronised DBS considers states with the same g-value, therefore these states can only have different f-values if they have different h-values.

than the cumulated costs of the states to be processed can be removed from memory. This resembles sweep-line state exploration [12].

4. In G-synchronised Bs variants with a monotonic G-function, bit-state hashing [33] can be used to reduce memory use. This technique is however inherently incomplete, i.e. may miss parts of the state space, and in particular when used in Bs there is the possibility of ignoring a previously visited state when it is reached via a path with a lower G-value. However, for G-synchronised Bs variants with monotonic G, this does not pose a problem, since re-opening of states is never needed. Note that the approach remains an approximation to Bs and, thus, can be seen as a trade-off between memory usage and having a tight grip on pruning.
5. In [46], a caching framework for both BFS and DFS was proposed. There, a hierarchy of state caches is employed to carefully keep a partial search history in memory, instead of the complete history $\bigcup_{j=0}^{i-1} \mathscr{L}_j$ (when being in round i). Each cache can keep a fixed number of elements, i.e. search levels (for BFS) or states (for DFS), has a sampling function determining which elements are accepted for storage, and has a replacement strategy determining which elements need to be removed, in case the cache is full. Moreover, the mechanism is adaptive, and still guarantees termination and exhaustiveness of the search algorithm. In practice, it was observed that memory requirements for state space generation can be reduced by at least 70%, sometimes even by more than 90%. This technique can be used with Bs as well, but one should be aware of the fact that the partial duplicate detection in such a setting may have a direct impact on the direction of the Bs, hence the quality of the Bs, as a state reintroduced for exploration, due to a failure in the duplicate detection, will compete for selection in the next round with other states. Reintroduced states can therefore be selected for exploration again, thereby pushing out states which otherwise would have been selected. Because of this, compared to performing a similar Bs without caching, a different subspace of the WLTS will be explored.

8. Heuristics and selecting the beam width

Effectiveness of Bs hinges on selecting good heuristic functions. Heuristic functions, as George Polya put it in 1945, are meant "to discover the solution to the present problem" [53], and thus heavily depend on the problem being solved. As the focus of this article is the development of search algorithms working with heuristics, we do not discuss techniques to design the heuristic functions themselves. Developing heuristics constitutes a whole separate body of research and, here, we refer to a few case studies on using heuristics in pruning state spaces: Among others, [30,47,59,64] present detailed discussions on pruning heuristics when dealing with Java program analysis, scheduling a wafer stepper machine, and jobshop scheduling problems, respectively. In Section 9, we show the effect of using heuristics to schedule some tasks in several applications, some based on river crossing problems [17].

Particularly papers on *designing* heuristic functions, such as the work by [21,19,30,39], constitute a nice complement to the work we present here, as they explain how to design heuristic functions and we start with the assumption of having a heuristic function. Authors [21,19] use guidelines to approximate the distance to deadlocks and violations of invariants and assertions. The objective of [30] is to model check Java programs with heuristics constructed using the properties to check, the structure of the programs and additional input of the user. Such functions can naturally be used as input also to the algorithms proposed in this article.

Selecting the beam width β is another challenge in using Bs. The beam width intuitively calibrates the time/memory usage of the algorithm on the one hand and the accuracy of the results on the other hand. Therefore, in practice the time/memory limits of a particular experiment determine β. To reduce the sensitivity of the results to the exact value of β, we propose using flexible Bs variants, cf. the results in Section 9. This, however, comes at the price of losing a tight grip on the memory consumption (see also Section 5.4).

For more general discussions on selecting β and its relation to the quality of answer we refer to [59].

9. Experiments

In this section, we first present a number of relatively small problems, which nevertheless nicely represent the class of problems the Bs variants discussed in this article are meant to be applied to. In particular, these problems produce WLTSs with interesting structures: they contain cycles, deadlocks (meaning unsuccessful terminations of attempts to solve the problem), and confluence of traces (i.e. there are states with multiple incoming transitions). Therefore, these problems show the effectiveness of our techniques to a great extent. We describe the problems, and report experimental results, which were obtained by using the μCRL toolset version 2.17.13. The results are analysed, and conclusions are drawn.

It should be stressed that the main targets for the search techniques are industrial case studies. In this section, some results are presented and discussed concerning the scheduling of an industrial system called the Clinical Chemical Analyser. In [67,69], this case is described in detail. Finally, we compare the efficiency and effectiveness of the μCRL Bs implementations and existing planning tools, using a number of planning problems. [7]

[7] For all material to do the experiments, see http://www.win.tue.nl/^awijs/suppls/bs.html.

Table 1
Experimental results C&M. Times are in hours:minutes:seconds. n.s., no solution exists [41]; o.o.t., out of time (set to 12 h)

Problem	Result	μCRL Mcs		μCRL g-SFDBS			
(C,B)	T	# States	Time	T	β	# States	Time
(3,2)	18	147	00:00:04	18	3	142	00:00:04
(10,3)	n.s.	396	00:00:04	n.s.	10	396	00:00:04
(10,4)	44	1378	00:00:04	46	10	1129	00:00:04
(20,4)	104	2537	00:00:05	106	10	2191	00:00:05
(50,10)	142	25,868	00:00:11	148	10	8035	00:00:08
(50,20)	116	90,355	00:00:20	120	15	17,361	00:00:11
(100,10)	292	49,141	00:00:20	296	10	16,274	00:00:14
(100,30)	222	366,608	00:01:06	228	15	61,380	00:00:32
(300,10)	892	143,549	00:01:02	896	10	49,514	00:00:48
(300,30)	680	1,008,436	00:04:11	684	15	205,556	00:02:30
(500,50)	1076	4,365,536	00:21:41	1080	20	685,293	00:10:33
(500,100)	1036	17,248,979	01:17:16	1040	20	1,170,242	00:16:47
(1000,50)	2160	8,551,996	01:10:00	2168	20	1,397,100	00:37:02
(1000,250)	o.o.t.	o.o.t.	o.o.t.	2032	20	5,317,561	04:00:22

9.1. Cannibals and missionaries

In this section, we report our experimental results on solving the *Cannibals and Missionaries* (CM) problem (see, e.g. [41]), which belongs to the class of *river crossing problems* [17]. We use a number of μCRL implementations of searches. First, we describe the problem. After that, we list the search techniques used and discuss the results, shown in Table 1.

9.1.1. Description of the problem

In the CM problem, C missionaries and C cannibals stand on the left bank of a river that they wish to cross, with $C \in \mathbb{N}$. There is a boat available which can ferry up to B people across ($B \in \mathbb{N}$). The goal is to find a schedule for ferrying all the cannibals and all the missionaries safely across, i.e. the cannibals never outnumber the missionaries, on a shore or in the boat. The boat can only move if it contains at least one person. On top of that we associate costs with moving the boat (1 time unit per passenger), and desire to find a minimal-cost path towards the goal.

9.1.2. Results

Table 1 provides all the obtained results. The experiments have been performed on a single machine with a 64 bit Athlon 2.2 GHz CPU and 1 GB RAM, running SUSE Linux 9.2, using the μCRL toolset version 2.17.13. The specifications and the commands used to invoke the searches can be found in [67].

In μCRL, we first applied the *minimal-cost* search, denoted Mcs in Table 1. Mcs is a Ucs applied on an LTS with additional actions representing the costs [67,69]. This search was used to find the minimum number of time units needed to solve the problem (shown in the *Result* column). The execution times of the searches are displayed in the corresponding *Time* column in the format 'hours:minutes: seconds'.

We also used g-synchronised flexible DBS (g-SFDBS), which is a combination of the techniques proposed in Sections 5.4 and 5.5, with $h(s) = C(s) + M(s) + (\langle C(s) \neq M(s)\rangle \times (2 \times C))$ as the heuristic part of the search, where $C(s)$ and $M(s)$ are the numbers of cannibals and missionaries on the left bank in state s, respectively, C is the total number of cannibals (or missionaries) in the problem, and $\langle C(s) \neq M(s)\rangle$ equals 1 if $C(s) \neq M(s)$, and 0 otherwise. The intuition behind this heuristic is that, first, we want to minimise $C(s)$ and $M(s)$. Second, we support having an equal number of cannibals and missionaries on the left bank as an easy way to avoid deadlock states where $C(s) > M(s)$, hence putting an extra penalty on such states.

Our experiments showed that in practice there are so many unsuccessful termination states in the specification that some deadlock avoidance in the heuristic function is needed. Without it, we often experienced unsuccessful searches in which the entire beam got trapped in deadlocks. With deadlock avoidance, flexible Bs proved to be applicable using a fairly stable beam width of 20, partially showing the suitability of the heuristics used. In Table 1, the $\boldsymbol{T}$ column under g-SFDBS shows the minimum number of time units needed to solve the problem approximated by this search. The results show an example of what can be achieved when near-optimal solutions are acceptable, i.e. when we give up completeness.

Let us take a closer look at a particular problem instance, using the 3D interactive visualisation tool LTSVIEW [43]. Fig. 8a shows us, on the left side, the complete WLTS of the (50,10) instance of the Cannibals and Missionaries problem, in other words, the case where there are 50 missionaries and 50 cannibals, and the boat can contain up to 10 people. The initial state is at the top of this structure. As it turns out, there is exactly one state representing successful termination, therefore all possible successful traces end up in this state. [8] The state is situated in the small cone near the bottom of the image, in the centre of

[8] The presence of only a single goal state seems to indicate that this WLTS is unsuitable for Bs. However, there are many traces leading to this state, so in fact, this state represents a large set of successful executions.

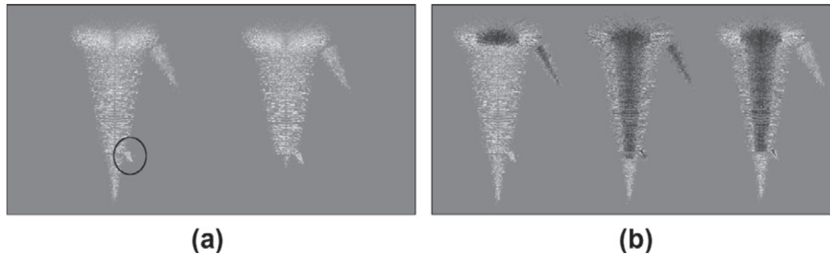

Fig. 8. The (50,10) CM problem. (a) BFS and MCS. (b) 1 – $\beta = 10$ without deadlock avoidance; 2 – $\beta = 100$ without deadlock avoidance; 3 – $\beta = 10$ with deadlock avoidance.

the black circle. When we search the WLTS with MCS, which is situated on the right of the figure, we observe that everything needs to be searched, except for the part at the bottom, which is at a greater depth than the cone containing the goal state.

If we consider the same problem instance using BS, we get the results shown in Fig. 8b. The importance of including some notion of deadlock avoidance in the estimation function becomes apparent here. On the left, we see the parts of the WLTS which are searched using a g-synchronised DBS (g-SDBS) with $\beta = 10$ and $h(s) = C(s) + M(s)$, displayed in dark grey. It is particularly interesting to note that, using this estimation function, the search quickly gets trapped in the 'deadlock cone' at the top, i.e. a part of the WLTS which only leads to failure. We can compensate for this behaviour by increasing the beam width. If we take $\beta = 100$, we find the goal state. This case is shown in the middle of the figure. In this case, however, it should be clear that we are not so successful at pruning. Besides, as it turns out in practice, the beam width needs to be increased in size considerably when dealing with increasingly big problem instances. On the right, we can see a g-SDBS with $\beta = 10$ and $h(s) = C(s) + M(s) + (\langle C(s) \neq M(s)\rangle \times (2 \times C))$. Now, with some form of deadlock avoidance in the estimation function, we are able to find the goal state with a beam width of 10, and furthermore are able to completely avoid the deadlock cone.

In this case study (as well as the one reported in the next section), we have chosen various values for β in order to demonstrate the effect the beam width has on the quality of search and the costs it incurs in terms of time and memory. Using the beam widths reported in Table 1 we observed a balance between the quality of the solutions and the resources used during the search. See Section 8 for discussions on choosing a value for β in general.

Our g-SFDBS algorithm should ideally be compared with other (heuristic) state space generation tools, such as HSF-SPIN, which is SPIN augmented by [20] with A^* and greedy BEFS. We leave this as future work. However, we have employed the SPIN DFS BNB algorithm of [57] to solve the C&M problem, which, on average, was less successful in finding schedules than BS. This is to be expected, though, as DFS BNB does not prune in the width, and is not an approximative search (at least, it exhaustively searches up to the depth of the optimal schedule).

9.2. The Zebra Finch problem

Next, we look at instances of what we call the *Zebra Finch problem*. We based this problem on a combination of several river crossing problems, such as *five jealous husbands* and *soldiers and children* [17]. First, we describe the problem, and then we provide the results obtained using the techniques described in this article, plus distributed versions, such that multiple machines can perform a BS together. For technical details concerning the distributed versions, see [67,68].

9.2.1. Description of the problem

Zebra Finches, *Taeniopygia guttata* [65], are small birds living in Central Australia [70] (Fig. 9). They are found in large colonies of pairs inhabiting open steppes with scattered bushes and trees. These birds can react aggressively towards each other, for instance when a jealous male bird tries to keep other male birds away from his mate. When young birds reach an age where they can live outside the nest, they are quickly adopted by the group.

We consider a group consisting of n pairs and m young, sitting in a tree on an open steppe. They want to migrate to some bushes up ahead, but they have to travel in smaller groups, since there are some hawks flying in the distance, which can spot a group of more than k adult finches. Once a group has reached the bushes, at least one of the Zebra Finches needs to fly back, in order to signal that a new group can travel. On top of this there are two other conditions:

1. Considering the jealous nature of the male Zebra Finches, no female finch may ever be either in the tree, the travelling group or the bushes in the presence of other male birds, unless her partner is also present.
2. The young in the colony have to be guided by at least one adult finch, so the travelling group cannot consist of only young finches. In limiting the group size, two young are equivalent to one adult.

Finally, some costs are related to the travelling from tree to bushes and back:

- A group consisting of only adults needs 1 time unit to travel the distance, independent of the size of the group;

Fig. 9. A pair of Zebra Finches.

Table 2
Zebra Finch problem experimental results. Times are in hours:minutes:seconds. o.o.m., out of memory (set to 900 MB); *, distributed search.

Instance			μCRL Mcs			μCRL g-Sdbs				μCRL g-Sfdbs			
n	m	k	Result	# States	Time	β	Result	# States	Time	β	Result	# States	Time
10	5	5	19	228,737	00:00:29	400	19	58,272	00:00:14	400	19	67,804	00:00:18
10	10	5	21	513,123	00:01:07	400	21	65,605	00:00:18	400	21	85,633	00:00:24
10	10	8	10	2,020,061	00:04:28	450	10	48,669	00:00:19	400	10	69,550	00:00:21
50	50	5	121	18,157,429	00:48:13	1000	121	641,315	00:04:49	400	121	298,065	00:02:31
50	50	10	41	475,744,120*	05:13:26	1000	43	637,285	00:07:28	1000	41	702,844	00:13:10
50	50	10	–	–	–	1500	44	946,660	00:13:37	1500	41	1,088,042	00:20:09
50	50	10	–	–	–	4000	43	2,560,051	00:46:22	4000	41	2,881,512	00:51:28
50	50	20	o.o.m.	o.o.m.	o.o.m.	5000	24	3,478,600	01:14:00	1500	22	1,521,829	00:54:07
50	50	20	o.o.m.	o.o.m.	o.o.m.	5000	20	3,095,782*	02:01:05	4000	20	2,579,479*	01:48:16
100	100	10	o.o.m.	o.o.m.	o.o.m.	5000	87	6,009,134*	01:39:52	4000	87	5,318,589*	06:22:54
100	100	20	o.o.m.	o.o.m.	o.o.m.	5000	41	5,884,895*	00:42:48	4000	42	5,433,733*	04:02:26
100	100	50	o.o.m.	o.o.m.	o.o.m.	20,000	17	27,366,213*	02:57:21	4000	18	41,611,293*	06:16:29
100	100	80	o.o.m.	o.o.m.	o.o.m.	20,000	10	24,796,756*	25:31:24	10,000	9	16,044,286*	05:05:49
200	200	50	o.o.m.	o.o.m.	o.o.m.	50,000	35	135,964,662*	37:25:42	6000	35	27,324,012*	08:32:35

- If the number of young in the group does not exceed the number of adults, the time needed to travel is 2 time units (each adult needs to take care of at most one young);
- When, in the group, the number of young exceeds the number of adults, the travel takes 3 time units, since at least one adult takes care of more than one young.

We specify the problem allowing all possible actions at all times. It demonstrates the techniques' ability to deal with arbitrary Wltss; problem instances lead to Wltss containing both cycles (while forming the group and when birds fly away and back again), and deadlocks (violations of the 'jealous male' condition).

9.2.2. Results

In Table 2, we present some results we found for instances of the Zebra Finch problem. We used Mcs, g-Sdbs, and its flexible variant (g-Sfdbs), where for the last two cases we defined $h(s)$ for each state s as the number of finches still in the tree, thereby encouraging fast removal and discouraging the returning of finches. Problem instances are described by providing n, m and k. For each search, the total cost of the result found is given as T. Furthermore, the number of states searched to find the solution is provided and the time needed to find it is displayed in the format 'hours:minutes:seconds'. When a search is done in a distributed setting, an asterisk is placed after the number of states. Sequential searches were performed using a machine with a 64bit Athlon 2.2Ghz processor, 1 GB of memory and running Suse 9.3, while 16 of these machines together performed the distributed searches.

With Mcs, as the problem instances get bigger, the Wltss grow very rapidly. The Bss on the other hand show a much nicer increase in states from instance to instance. Looking at the (50,50,10) instance though, we see an unwanted effect in the

Table 3
Experimental results Cca. Times are in hours:minutes:seconds. o.o.t., out of time (set to 30 h); *, distributed search.

Case	Result	μCRL Mcs		μCRL g-Sdbs			μCRL g-Spbs			μCRL g-Sfpbs	
		# States	Time	β	# States	Time	(α, l)	# States	Time	# States	Time
(3,1,1)	36	3375	00:00:10	25	1461	00:00:03	(2,5)	179	00:00:04	821	00:00:04
(1,3,1)	39	13,194	00:00:30	41	2234	00:00:04	(1,1)	50	00:00:03	1133	00:00:04
(6,2,2)	51	341,704,322*	25:24:56	81	7408	00:00:08	(2,9)	479	00:00:03	45,402	00:02:34
(1,2,7)	73	o.o.t.	o.o.t.	75,000	6,708,705	01:24:38	(1,1)	90	00:00:03	122,449	00:04:03
(7,4,4)	75	o.o.t.	o.o.t.	35,000	3,801,607	00:41:02	(3,25)	155,379	00:08:15	20,666,509	14:32:56

regular g-Sdbs, namely that increasing β not necessarily means getting a better result. The main cause for this is tie-breaking, i.e. the fact that pruning is sometimes not being done only based on f, but also on other criteria, simply because more than β states turn out to be promising enough. Although this mainly has a noticeable effect in smaller instances, it is undesired and does not occur in its flexible variant. Because of this, the flexible search provides better insight into the effectiveness of the estimation function used.

Furthermore, it is interesting that for smaller instances, the distributed algorithm performs worse than the sequential version, which can be seen in the (50,50,20) case, where we performed both a sequential and a distributed search. The $\mathscr{L}_i$ sets in the Wlts are all relatively small, making the communication overhead of the distributed algorithm noticeable. This seems to be directly related to the argument found in the literature, for instance by [7], against distributed Bs in a more traditional setting. Besides that, note that the result obtained with the distributed search is better than the one of the sequential search, even though the beam widths are equal. This, again, is due to tie-breaking, which, in a distributed environment, can happen at multiple places in a single level, instead of only at one point. In the flexible search, where tie-breaking is avoided altogether, this behaviour does not appear.

The (100,100,50) and the (100,100,80) case have a big difference in execution time, while the number of states in the latter is even smaller. However, although the number of expanded states is smaller there, the number of encountered and evaluated states is much greater. This is directly related to the maximum size of the travelling group k.

Finally, as stated earlier in Section 5.4, for the flexible search, overall β is more stable compared to the non-flexible search. Due to this, in the two biggest cases, the flexible search is even more efficient than the non-flexible one. The stability of β means in general that, given some search results, it is easier to determine β for a new flexible search than for a new non-flexible one.

9.3. Other benchmarks

The *Clinical Chemical Analyser* (Cca) is a case study taken from industry [66]: it is used to analyse batches of test receipts on patient samples, such as blood and plasma, that are uniquely described by a triple which indicates the number of samples of each fluid (see Table 3). We have extensively described the Cca case in [69]. Table 3 reports the results of applying Mcs, g-Sdbs, g-Spbs and g-Sfpbs to solve the problem of scheduling the Cca. The Result column provides the total-cost (i.e. required time units) of the solution found. We remark that all these searches are tuned to find the optimal answer (for those cases where it was known to us). In case of g-Sfpbs, the value of (α, l) is fixed to $(1, 1)$. The benefit of flexible variants of Bs is thus clear here: A stable beam width is mostly sufficient. However, as a draw-back we observe that Fpbs exhibits early state space explosion, compared to Pbs. [9]

We observe that β is not directly related to the number of fluids in a test case. We believe this can be due to the ordering of states while searching, since a stable β suffices when using the flexible Sfpbs. We conclude this discussion with noting that the Cca provides a case study which can better be tackled using Pbs, compared to Dbs variants.

Finally, we have taken some planning benchmarks from the Beem database [10] and translated these to μCRL specifications. Two of these, *blocks world* (BW) and *schedule world* (SW), were also used in the AIPS 2000 planning competition. [11] We are therefore able to compare our Bs results for several instances of these problems with the ones obtained in the planning competition (although it should be noted that the Beem version of SW is a slightly modified variant of the AIPS problem). A third problem we analysed is the *Exit* game, instance 4, where each of a finite number of locations in a city contains a number of puzzles to solve. Some of the puzzles can only be solved in some specified time periods, and several teams must compete in solving a given number of puzzles (in this case 8 out of 15) and moving to the finish. The goal of the problem is to find a schedule such that *all* teams are finished as quickly as possible. With μCRL Mcs, a solution with length 51 was found after exploration of $35,643,844$ states, which took 4 h, 34 min, and 27 s, while μCRL g-Sfdbs with $\beta = 1$ produced a result of equal length, after exploration of $3,966$ states, which took 2 s. For all three problems, we used abstract data types to express a heuristic guiding function in μCRL; for *Exit*, the function is based on a simplified, untimed, version of the problem. For BW, in which a finite number of blocks on a table needs to be placed on a stack in a specific order, we use the well-known global heuristic which compares the sizes of correct substacks with those of incorrect substacks. Finally,

[9] In Fpbs, once the beam width is stretched, it cannot be readjusted to its initial value, see Section 5.4.
[10] http://anna.fi.muni.cz/models, visited 11 February 2011.
[11] http://www.cs.toronto.edu/aips2000., visited 11 February 2011.

Table 4
Experimental results for several planning problems from AIPS 2000. Times are in hours:minutes:seconds. n.a., not available.

Case	μCRL g-SFDBS ($\beta = 1$)			HSP2		PBR	
	Result	# States	Time	Result	Time	Result	Time
BW10-0	34	413	00:00:14	34	00:00:12	34	00:00:00
BW13-0	44	896	00:01:05	72	00:00:22	42	00:00:00
BW16-1	56	4739	00:09:23	n.a.	n.a.	56	00:00:00
BW30-0	104	14,501	03:04:57	n.a.	n.a.	98	00:00:02
SW8-0	11	341	00:00:01	11	00:00:25	10	00:00:05
SW10-0	18	11,968	00:00:08	n.a.	n.a.	15	00:00:10
SW25-0	29	4215	00:04:00	n.a.	n.a.	39	00:02:20
SW50-0	68	22,136	01:19:47	n.a.	n.a.	87	00:14:22

for SW, in which a number of products need to be processed in a specific way by a number of machines, a guiding function was designed which stimulates to perform as many productive tasks in parallel as possible. Table 4 shows a comparison of results for several instances of BW and SW obtained with g-SFDBS, with $\beta = 1$, and with the planners *Heuristic Search Planner 2* (HSP2) [10] and *Planning by Rewriting* (PBR) [3]. In the AIPS 2000 competition, there were two planner categories, consisting of automatic planners and hand tailored planners. Planners of the latter category are allowed some manual input in addition to the problem specification, e.g. a guiding heuristic, while the planners in the former category are not. HSP2, an automatic planner, performed very well in the competition. It uses a weighted A^* search through a state space representing the problem, and guiding heuristics are extracted from the representation. PBR is a hand tailored planner, which generates plans by a set of plan rewriting rules, where some rules are automatically generated and others are given manually.

We remark that it is in general very difficult to compare these planning tools with the μCRL toolset. This is because, firstly, there is a considerable difference between the models used in planners and μCRL models. Secondly, the μCRL toolset has a general-purpose state space generation tool, while the planners are geared towards solving planning problems. That is, the expressive power of μCRL for modelling case studies, along with the possibility of performing qualitative model checking based on μCRL models, gives the μCRL toolset an advantage over the planners. However, one would expect that the planners outperform the μCRL toolset in terms of efficiency of searching for optimal plannings. There are other sources of complications in comparing the μCRL toolset with the planners. For instance, the μCRL algorithms do not automatically extract heuristics from a specification, as opposed to HSP2. Moreover, the experiments were run on different architectures: the μCRL experiments were performed on a 2.6 GHz AMD OPTERON 885 with 126 GB RAM running LINUX, and the planner experiments were run on a 500Mhz Pentium III with 1GB RAM running LINUX.

Nevertheless, meaningful conclusions can be drawn from Table 4. Firstly, the quality of the results (expressed by the number of steps in the suggested plan) with g-SFDBS is on average high. Secondly, the time needed to generate the plans with g-SFDBS is often comparable to the time needed by the planners (in the SW cases even with PBR, which also depends on additional manual input). We expect that the planners outperform our algorithms when run on the same architecture, but it should be noted that apart from the BS implementation, the μCRL toolset is not tailored towards solving planning problems. From Table 4, we conclude that model checkers, in particular the μCRL toolset, can be used for solving planning problems. The quality of the solutions are (favourably) comparable to the results produced by the tools which are devoted to planning. The benefits of using μCRL for this purpose is the possibility to simultaneously perform qualitative model checking on the same models that are used to solve the planning problem.

10. Related work

10.1. Other uses of beam search

The literature on traditional BS mainly focuses on how BS is useful for solving a specific problem and no general framework is presented. E.g. in [63], a relatively small specification of a typical jobshop problem is provided that does not include data structures, which are often necessary when dealing with practical systems. In [47], a very specific program is used, which is only able to simulate the case study presented there. In these papers, BS is used on a case by case basis, therefore reusing their implementation of BS on other case studies is not straightforward. We provide a general framework, based on an expressive specification language, instead of case-based tools. This allows us to easily describe complex systems and various problem restrictions.

In many applications of BS, no restrictions, neither timing nor data, are initially put on scheduling jobs (e.g. in simple jobshop scheduling problems [50] or in the case of [47]). However, in single machine early/ tardy jobshop scheduling problems [63,64], for instance, there are (timing) restrictions on scheduling jobs. But the violation of these restrictions is usually allowed while penalties are put on them, hence not excluding violations from the search space. These restrictions, in their most general form, can either be hard, meaning that they have to be necessarily met, or soft, that is the violation of these requirements will result in a penalty, but is still allowed. Soft restrictions can simply be modelled by adding extra costs on prohibited actions. We contend that hard restrictions should be specified in the model, because allowing unwanted

executions leads to a search space larger than necessary. This, however, requires an expressive specification language, which seems not readily available, e.g. for [63,64], where restrictions are applied on the model after generation. In μCRL, conditions on data and timing restrictions can be specified in a straightforward manner.

The flexible variants of Bs presented in this article are remotely similar to Bs *with variable width*, as described by Valente and Alves [63]. They completely leave out a predefined beam width and introduce deviation parameters so that, broadly speaking, the algorithm calculates how far the evaluation of a node may be from the optimal evaluation value of that level to still be selected for exploration. In comparison, Valente and Alves take away some influence of the evaluation function, to be able to consider more possibilities when searching a tree, whereas we give more influence to the evaluation function and do not allow any other criteria to affect the selection, in order to reestablish the importance of the evaluation function after we moved the searches to the Wlts setting. There are significant implementation differences between flexible Bs and Bs with variable width of [63]. In Dbs with variable width, at each level, the largest total-cost value of the states of the level must be known before selection can proceed. This completely disables the second optimisation mentioned in Section 5.3. Furthermore, in Pbs with variable width, the priority threshold has to be separately computed for each node, which can be computationally expensive.

Our g-synchronised Bs can probably best be compared with filtered Bs [59], since both are two-phase Bss; in filtered Bs, first a Pbs is applied, and on the outcome of that, Dbs is used, this to lessen the computational complexity. In g-synchronised Bs, we first postpone exploring some states, and then prune states from the remaining set. Both searches can be seen as instances of m-Befs (Section 3).

A number of search algorithms is used in [30], one of which is called Bs. Their Bs, however, deviates from our usage, in that they let $f(s) = h(s)$, making it practically a linear space greedy search; it is, nevertheless, according to the original notion, a Bs. Furthermore, they include duplicate detection, but do not consider other extensions in order to deal more efficiently with arbitrary Wltss, such as a flexible beam width.

Bs is extended to a complete search in [72], by using a new data structure, called a *beam stack*. With this, it is possible to achieve a range of searches, from Dfs ($\beta = 1$) to Bfs ($\beta = \infty$). Considering our extensions for arbitrary Wltss, it would be interesting to try to combine these two approaches. *Iterative broadening* [28] and *iterative widening* [71] are two other closely related approaches to make Bs complete. Essentially, these iterative searches incorporate applying Bs multiple times, each time increasing β. As noted by [72], although this means that these searches are complete, they are only so because the final iteration will exhaustively search the entire Wlts. Hence, worst-case, they require that the entire Wlts can be stored in memory. In [18], a search width k is applied on Bfs BnB for priced timed automata. The resulting algorithm is made complete by using iterative broadening. There, k expresses the percentage of states that should be explored per level of the Wlts; they also allow the selection of more than k%, which is comparable to our flexible beam width. The usage of a heuristic function is not considered; instead, selection is done based on cumulated costs.

10.2. Connections to other heuristic search algorithms

We observe that the Por algorithm of [14] for security protocols can be seen as an instance of flexible Pbs. The main principle of Por is to exploit the commutativity of concurrently executed transitions in order to generate only a sufficient fraction of the state space by exploring a subset of enabled transitions $ample(s) \subseteq en_{\mathcal{M}}(s)$ at each state s. This resembles Pbs, since at each state, based on the suitability of the enabled transitions, some of the successors are pruned away while generating. However, in contrast to Pbs, no essential information is lost in Por as the *ample* set is selected such that a certain class of desired properties is preserved. We refer to [13,49] for a general introduction to Por. In [62], a translation from this algorithm to the general pruning framework is provided, and in [25], the algorithm within this framework is extended to be applicable to branching security protocols.

In [24], Befs is extended to k-Befs, allowing to compensate for inaccuracies in the evaluation function by selecting in each iteration more than only the best state. Essentially, the difference between k-Befs and Bs is the decision to keep states not selected in one iteration for the next iteration, i.e. to not prune away any states. This makes k-Befs a complete search, but it also means its memory requirement is higher. A trade-off can, however, be achieved, by using inadmissible heuristics, such that fewer states are expanded, but the solution will be near-optimal. This trade-off is also used for weighted A^* [52], and linear-space Befs [37], where the h-function is multiplied by some factor. Besides that, in the latter, the memory requirement is linear to the size of the search depth.

Iterative deepening A^* (IDA^*) [35] performs multiple depth-bounded A^* searches, each time increasing the depth bound. Since our detailed Bs algorithms also employ an evaluation function $f(s) = g(s) + h(s)$, the only difference between these algorithms and a single iteration of IDA^* lies in the way they prune; the latter prunes in the depth, while the former prune in the width. The effect of this is that an iteration of IDA^* is complete up to the depth bound, while Bs is not, but it can reach great depths much faster. In total, IDA^* is complete, but it suffers from the same drawback as iterative broadening (see Section 10.1).

Our work is related to the body of research on Dmc, where, to find a counter-example to a functional property (usually belonging to Ltl) with a minimal exploration of the Wlts, heuristics (based on the property) are used to guide the search. Using A^* [19] and genetic algorithms [29] to guide the search are among notable works in this field. In contrast to Dmc, we generate a *partial* Wlts in which an arbitrary property can be checked *afterwards* (the result would of course not be exact, hence being useful mainly in quantitative analyses where a near-optimal solution for a problem often suffices). In this

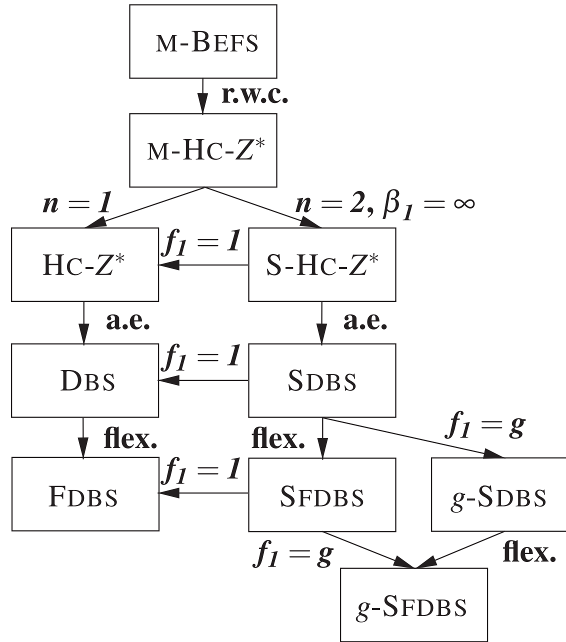

Fig. 10. Hierarchical diagram of searches; r.w.c.: recursive-weight-computation [48], a.e.: additive evaluation [48], flex.: flexible β.

sense, these approaches are different in spirit, addressing rather different problems (i.e. checking qualitative vs. quantitative properties) from different angles (directing the search vs. searching an approximation to the WLTS). Nonetheless, there are strong similarities as well: the approach of [29] is similar to ours, as it is in general not guaranteed to explore the whole WLTS, and A^* can be seen as an instantiation of our BS extensions. Related to the latter remark, in [48], a hierarchical diagram is given in which A^* is considered a special case of BEFS. More specifically, A^* is a special case of Z^*, which is a subclass of BEFS consisting of searches which have *delayed-termination*, i.e. termination is detected once a goal state is selected for exploration, and *recursive-weight-computation*, meaning that for each state, a weight is computed, which is defined recursively. BEFS in [48] does not incorporate irrevocable pruning, though. Instead, adding this option leads to a hybrid called HC-BEFS, where HC stands for Hill-Climbing. Fig. 10 presents a diagram similar to the one in [48], in which the relations between M-BEFS and the DBS extensions are shown. A connection with [48] can be made as follows: our BEFS, which is a special case of M-BEFS, corresponds with HC-BF* in [48]. By requiring that recursive-weight-computation is done, M-BEFS can be tuned to M-HC-Z^*, a multi-phase version with pruning of Z^*. Fixing the number of phases n can lead to HC-Z^* (1-phase) and Synchronised HC-Z^* (2-phase). Of course, variants of HC-Z^* with more phases are feasible as well. By requiring *additive evaluation* [48], which means that some notion of cumulated cost is computed, we can obtain DBS with 1 or 2 phases. By making β flexible, we get flexible BS variants. If f_1 (the evaluation function used in the first phase) is made constant, e.g. $f_1 = 1$, then a synchronised search is changed into a 1-phase search, and if $f_1 = g$, then g-synchronised BSS are created. A similar diagram can be drawn for PBS-variants; in that case, instead of additive evaluation, priority evaluation should be used (which is more local). When moving to g-synchronised BSS, though, additive evaluation is still added, since a cumulated cost computation must then also be performed.

10.3. Applicability in other quantitative model checking areas

In this article, BS has exclusively been applied on specifications of scheduling problems. However, since BS can be applied on general WLTSS, the technique is also applicable on other types of (quantitative) problems. In this section, a few of these are briefly discussed.

First of all, basic (unweighted) model checking [13] can be seen as a special kind of quantitative model checking, where all transitions have cost 1. In model checking, the goal is to check whether a specification of a system, which implicitly describes an LTS containing all potential behaviour of the system, meets a set of requirements. These requirements are expressed formally in temporal logics formulae, using, e.g. regular alternation-free μ-calculus [45]. Many requirements are so-called *safety properties*, which express that some bad situation never happens. Such a formula can be mapped to a subset of $\mathscr{S}$, i.e. $\mathscr{G}$, which technically represents counter-examples to the property. Hence, in model checking, we are interested in 'bad' situations, and how these can occur, which is represented by the traces from $\mathscr{I}$ to $\mathscr{G}$. Since short traces are easier to interpret, BFS is a popular search for model checking (note that when all transitions have cost 1, UCS coincides with BFS). In order to perform BS, we need an estimation function. In, e.g. [22,30,44], several methods are described to compute these.

In *stochastic* model checking, probabilities are assigned to the transitions, indicating the probability that the represented behaviour occurs. A common question in that context is not only if some bad situation can occur, but what the probability is that it occurs. The cumulated cost, or probability, of a state via a specific trace can be computed by multiplying all probabilities of the included transitions, which means that this function is not monotonic. Furthermore, typically, one is interested in the sum of the cumulated costs of *all* traces to a bad state [22], as opposed to the minimal cumulated cost associated with a single trace. This makes that Ucs is not very useful here. In [2], an algorithm is described to iteratively produce a Markov Chain [12] containing the set of all traces from $\mathscr{I}$ to $\mathscr{G}$. This algorithm searches for bad traces on-the-fly by using an evaluation function $f(s) = g(s) \times h(s)$, with g(s) the cumulated probability of s, and $h(s)$ an estimate of the probability to reach a bad state from s. Note that with such a function, the states with the highest f-value are the most interesting. Suggestions how to construct an estimation function are given in [1]. If we would apply pruning on the algorithm of [2], we would get a Bs, which could be used to generate a subset of the set of all bad traces. Depending on the quality of the heuristics, this subset would contain those bad traces with the highest cumulated probability.

Real-time model checking involves some notion of time to verify timed systems. Dmc techniques can be applied in this setting as well, as shown by, e.g. [5,6,39]. Bs could also be applied in this setting without any significant changes. If one wants to incorporate the timing itself in the guiding functions, and the timing is continuous, then it should be represented by some discrete cost. For hybrid systems, in which multiple variables may represent continuous behaviour, [51] describes a technique combining model checking and motion planning; Ucs is used on a discrete representation of the hybrid system behaviour, and the intermediate results of this are used by the motion planner to build a tree describing part of the hybrid behaviour. Applying pruning in combination with this technique would yield a Bs suitable for approximately checking safety properties of hybrid systems.

11. Conclusions

In this article, we extended and made available an existing search technique to be used for quantitative analysis within a setting used for system verification. By doing so, we contribute to attempts to achieve a general framework in which different types of analysis, such as functional verification and scheduling, can be performed on a single system specification. Moving Bs to the field of state spaces, we experienced that the algorithms needed to be extended in order to counter a decrease of influence of the heuristic function used. Bs with a flexible beam width can cope with encountering more than β sufficiently promising states. Using the algorithm for generating Wltss also introduces the well-known issue of re-opening states in Dmc. G-synchronised Bs is another introduced extension of Bs, of which the instance g-synchronised Bs, in cases where the g-function is monotonic, avoids re-opening states, by using a two-phase approach; in the first phase, states are ordered based on the cost needed to reach them from $\mathscr{I}$. In the second phase, the states are considered in this order and an estimation function is applied to prune relatively unpromising states. This extension can be identified, among some other searches found in the literature, as a particular instance of a Befs not previously discussed in the literature. We propose to call this extension of Befs m-Befs. In m-Befs, several search criteria can be 'stacked' by using multiple guiding functions; given a search horizon, these functions are applied in sequence, leading to a final selection of states to be explored.

The modelling language μCRL is well-suited for modelling scheduling problems. The data support it has is very convenient when working with complex data structures. In this regard, no changes had to be made to the current μCRL toolset. In other regards, the μCRL toolset had to be extended with search algorithms other than Bfs. Although not a necessity, a useful feature in the modelling language μCRL would be a priority operator, which could be used to assign priorities to actions.

We showed that g-synchronised (flexible) Bs is suitable for finding near-optimal solutions for instances of several river crossing problems and planning problems, as well as for an industrial case study. In these particular problems, the Wltss incorporate cycles, confluence of traces, and unsuccessful termination states, thereby they are useful examples to demonstrate that the Bs variants presented in this article can deal with arbitrary Wltss. Bs allows one to make a trade-off between computation time and the quality of the solutions to find. Having both Dbs and Pbs to work with, even increases the possibilities for such a trade-off. If one wants a certain level of quality, however, choosing the right beam width becomes a problem.

Because of this, in this article, we proposed extensions of both Dbs and Pbs, called flexible Bs, in which the actual beam width can change while searching, in order to keep track of all actions with a sufficient priority in each level (i.e. avoiding tie-breaking). The experiments suggest that from case to case, the beam widths of flexible Bss do not have to be increased often. The major benefits of flexible Bss are the relative 'stability' of the beam widths (i.e. when increasing the size of the test configuration, the beam width can often be left unchanged) and the avoidance of tie-breaking, but this comes at a price, namely that the space and computation time requirements of these searches are not linear to the maximum search depth.

Acknowledgements

We thank the anonymous reviewers of the Journal of Logic and Algebraic Programming for their constructive comments.

[12] Markov Chains are more related to weighted Kripke structures, in which the states instead of the transitions are labelled, than to Wltss. However, after some minor changes, the algorithm of [2] could also be applied on Wltss.

References

[1] H. Aljazzar, H. Hermanns, S. Leue, Counterexamples for timed probabilistic reachability, Proc. FORMATS 2005, LNCS, vol. 3829, Springer-Verlag, Heidelberg, 2005, pp. 177–195.
[2] H. Aljazzar, S. Leue, Directed explicit state-space search in the generation of counterexamples for stochastic model checking, IEEE Trans. Softw. Eng. 36 (1) (2010) 37–60.
[3] J.L. Ambite, C.A. Knoblock, Planning by rewriting, J. Artif. Intell. Res. 15 (1) (2001) 207–261.
[4] G. Behrmann, A. David, K.G. Larsen, A tutorial on Uppaal, Proc. SFM-RT 2004, LNCS, vol. 3185, Springer-Verlag, Heidelberg, 2004, pp. 200–236.
[5] G. Behrmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersson, J.M.T. Romijn, Efficient guiding towards cost-optimality in Uppaal, Proc. TACAS 2001, LNCS, vol. 2031, Springer-Verlag, Heidelberg, 2001, pp. 174–188.
[6] G. Behrmann, K.G. Larsen, J.I. Rasmussen, Optimal scheduling using priced timed automata, SIGMETRICS Perform. Eval. Rev. 32 (4) (2005) 34–40.
[7] R. Bisiani, Beam search, Encyclopedia of Artificial Intelligence, Wiley, 1992, pp. 1467–1568.
[8] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. van Langevelde, B. Lisser, J.C. van de Pol, μCRL: a toolset for analysing algebraic specifications, Proc. CAV 2001, LNCS, vol. 2102, Springer-Verlag, Heidelberg, 2001, pp. 250–254.
[9] S.C.C. Blom, J.C. van de Pol, M. Weber, LTSmin: distributed and symbolic reachability, Proc. CAV 2010, LNCS, vol. 6174, Springer-Verlag, Heidelberg, 2010, pp. 354–359.
[10] B. Bonet, H. Geffner, Heuristic search planner Ver. 2.0, AI Mag. 22 (3) (2001) 77–80.
[11] E. Brinksma, H. Hermanns, J.P. Katoen (Eds.), Lectures on Formal Methods and Performance Analysis, LNCS, vol. 2090, Springer-Verlag, Heidelberg, 2001
[12] S. Christensen, L.M. Kristensen, T. Mailund, A sweep-line method for state space exploration, Proc. TACAS 2001, LNCS, vol. 2031, Springer-Verlag, Heidelberg, 2001, pp. 450–464.
[13] E.M. Clarke, O. Grumberg, D.A. Peled, Model Checking, The MIT Press, 1999.
[14] E.M. Clarke, S. Jha, W. Marrero, Partial order reductions for security protocol verification, Proc. TACAS 2000, LNCS, vol. 1785, Springer-Verlag, Heidelberg, 2000, pp. 503–518.
[15] F. Della Croce, V. T'kindt, A recovering beam search algorithm for the one-machine dynamic total completion time scheduling problem, J. Oper. Res. Soc. 53 (11) (2002) 1275–1280.
[16] E.W. Dijkstra, A note on two problems in connection with graphs, Numer. Math. 1 (1959) 269–271.
[17] H.E. Dudeney, Amusements in Mathematics, Dover Publications Inc., 1958., pp. 12–114 (Chapter 9).
[18] S. Edelkamp, S. Jabbar, Real-time model checking on secondary storage, Proc. MoChArt 2006, LNAI, vol. 4428, Springer-Verlag, Heidelberg, 2007, pp. 68–84.
[19] S. Edelkamp, S. Leue, A. Lluch-Lafuente, Directed explicit-state model checking in the validation of communication protocols, Int. J. Softw. Tools Technol. Transfer 5 (2) (2004) 247–267.
[20] S. Edelkamp, A. Lluch-Lafuente, S. Leue, Directed explicit model checking with HSF-Spin, Proc. SPIN 2001, LNCS, vol. 2057, Springer-Verlag, Heidelberg, 2001, pp. 57–79.
[21] S. Edelkamp, A. Lluch-Lafuente, S. Leue, Protocol verification with heuristic search, AAAI Symposium on Model-Based Validation of Intelligence, AAAI, 2001, pp. 84–92.
[22] S. Edelkamp, V. Schuppan, D. Bošnački, A.J. Wijs, A. Fehnker, H. Aljazzar, Survey on directed model checking, Proc. MoChArt 2008, LNAI, vol. 5348, Springer-Verlag, Heidelberg, 2009, pp. 65–89.
[23] K. Etessami, M. Kwiatkowska, M.Y. Vardi, M. Yannakakis, Multi-objective model checking of Markov decision processes, Proc. TACAS 2007, LNCS, vol. 4424, Springer-Verlag, Heidelberg, 2007, pp. 50–65.
[24] A. Felner, S. Kraus, R.E. Korf, KBFS: K-Best-First Search, Ann. Math. Artif. Intell. 39 (1–2) (2003) 19–39.
[25] W.J. Fokkink, M. Torabi Dashti, A.J. Wijs, Partial order reduction for branching security protocols, Proc. ACSD 2010, IEEE Computer Society Press, 2010, pp. 191–200.
[26] M.S. Fox, Constraint-directed search: a case study of job-shop scheduling, Ph.D. Thesis, Carnegie-Mellon University, 1983.
[27] H. Garavel, F. Lang, R. Mateescu, An overview of Cadp 2001, European Association for Software Science and Technology (EASST) Newsletter, vol. 4, 2002, pp. 13–24.
[28] M.L. Ginsberg, W.D. Harvey, Iterative broadening, Artif. Intell. 55 (2) (1992) 367–383.
[29] P. Godefroid, S. Khurshid, Exploring very large state spaces using genetic algorithms, Proc. TACAS 2002, LNCS, vol. 2280, Springer-Verlag, Heidelberg, 2002, pp. 266–280.
[30] A. Groce, W. Visser, Heuristics for model checking Java programs, Int. J. Softw. Tools Technol. Transfer 6 (4) (2004) 260–276.
[31] J.F. Groote, J. Keiren, A. Mathijssen, B. Ploeger, F. Stappers, C. Tankink, Y.S. Usenko, M. van Weerdenburg, W. Wesselink, T.A.C. Willemse, J. van der Wulp, The mCRL2 Toolset, in: Proc. WASDeTT 2008, 2008, pp. 5–1/10.
[32] J.F. Groote, A. Ponse, The syntax and semantics of μCRL, Proc. ACP 1994, Workshops in Computing Series, Springer-Verlag, Heidelberg, 1995, pp. 26–62.
[33] G.J. Holzmann, An analysis of bitstate hashing, Formal Methods Syst. Des. 13 (3) (1998) 289–307.
[34] G.J. Holzmann, The Spin Model Checker: Primer and Reference Manual, Addison-Wesley, 2004.
[35] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, Artif. Intell. 27 (1) (1985) 97–109.
[36] R.E. Korf, Uniform-cost search, in: S. Shapiro (Ed.), Encyclopedia of Artificial Intelligence, Wiley, 1992, pp. 1461–1462.
[37] R.E. Korf, Linear-space best-first search, Artif. Intell. 62 (1) (1993) 41–78.
[38] V. Kumar, Branch-and-bound search, in: S. Shapiro (Ed.), Encyclopedia of Artificial Intelligence, Wiley, 1992, pp. 1468–1472.
[39] S. Kupferschmid, J. Hoffmann, H. Dierks, G. Behrmann, Adapting an AI planning heuristic for directed model checking, Proc. SPIN 2006, LNCS, vol. 3925, Springer-Verlag, Heidelberg, 2006, pp. 35–52.
[40] K. Larsen, J. Rasmussen, Optimal conditional reachability for multi-priced timed automata, Proc. FOSSACS 2005, LNCS, vol. 3441, Springer-Verlag, Heidelberg, 2005, pp. 234–249.
[41] R. Lim, Cannibals and missionaries, Proc. APL 1992, ACM Press, 1992, pp. 135–142.
[42] B.T. Lowerre, The HARPY speech recognition system, Ph.D. Thesis, Carnegie-Mellon University, 1976.
[43] LTSView, 3D Interactive Visualisation of a State Space, 2007. Available from: <http://www.mcrl2.org> (accessed 10.05.07).
[44] A. Lluch-Lafuente, Directed search for the verification of communication protocols, Ph.D. Thesis, University of Freiburg, 2003.
[45] R. Mateescu, M. Sighireanu, Efficient on-the-fly model-checking for regular alternation-free mu-calculus, Sci. Comput. Programming 46 (3) (2003) 255–281.
[46] R. Mateescu, A.J. Wijs, Hierarchical adaptive state space caching based on level sampling, Proc. TACAS 2009, LNCS, vol. 5505, Springer-Verlag, Heidelberg, 2009, pp. 215–229.
[47] S. Oechsner, O. Rose, Scheduling cluster tools using filtered beam search and recipe comparison, Proc. 2005 Winter Simulation Conference, IEEE Computer Society Press, 2005, pp. 2203–2210.
[48] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison-Wesley, 1984.
[49] D. Peled, V. Pratt, G. Holzmann (Eds.), Partial order methods in verification, Series in Discrete Mathematics and Theoretical Computer Science, vol. 29, American Mathematical Society, 1996
[50] M. Pinedo, Scheduling: Theory, Algorithms, and Systems, Prentice-Hall, 1995.
[51] E. Plaku, L.E. Kavraki, M.Y. Vardi, Falsification of LTL safety properties in hybrid systems, Proc. TACAS 2009, LNCS, vol. 5505, Springer-Verlag, Heidelberg, 2009, pp. 368–382.
[52] I. Pohl, Heuristic search viewed as path finding in a graph, Artif. Intell. 1 (3) (1970) 193–204.
[53] G. Polya, How to Solve it, second ed., Princeton University Press, 1945.
[54] D. Poole, A. Mackworth, R. Goebel, Computational Intelligence: A Logical Approach, Oxford University Press, 1998.

[55] S. Rubin, The ARGOS image understanding system, Ph.D. Thesis, Carnegie-Mellon University, 1978.
[56] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice-Hall, 1995.
[57] T.C. Ruys, Optimal scheduling using Branch-and-Bound with SPIN 4.0, Proc. SPIN 2003, LNCS, vol. 2648, Springer-Verlag, Heidelberg, 2003, pp. 1–17.
[58] I. Sabuncuoglu, M. Bayiz, Job shop scheduling with beam search, Eur. J. Oper. Res. 118 (2) (1999) 390–412.
[59] P. Si Ow, E.T. Morton, Filtered beam search in scheduling, Int. J. Prod. Res. 26 (1) (1988) 35–62.
[60] P. Si Ow, E.T. Morton, The single machine early/tardy problem, Manag. Sci. 35 (2) (1989) 177–191.
[61] P. Si Ow, S.F. Smith, Viewing scheduling as an opportunistic problem-solving process, Ann. Oper. Res. 12 (1–4) (1988) 85–108.
[62] M. Torabi Dashti, A.J. Wijs, Pruning state spaces with extended beam search, Proc. ATVA 2007, LNCS, vol. 4762, Springer-Verlag, Heidelberg, 2007, pp. 543–552.
[63] J.M.S. Valente, R.A.F.S. Alves, Beam search algorithms for the single machine total weighted tardiness scheduling problem with sequence-dependent setups, Comput. Oper. Res. 35 (7) (2008) 2388–2405.
[64] J.M.S. Valente, R.A.F.S. Alves, Filtered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time, Comput. Ind. Eng. 48 (2) (2005) 363–375.
[65] L.J.P. Vieillot, Taeniopygia. guttata, Abel Lanoe, 1817.
[66] S. Weber, Design of real-time supervisory control systems, Ph.D. Thesis, Eindhoven University of Technology, 2003.
[67] A.J. Wijs, What to do next? Analysing and optimising system behaviour in time, Ph.D. Thesis, Vrije Universiteit Amsterdam, 2007.
[68] A.J. Wijs, B. Lisser, Distributed extended beam search for quantitative model checking, vol. 4428Proc. MoChArt 2006, LNAI, vol. 12, Springer-Verlag, Heidelberg, 2007, pp. 165–182.
[69] A.J. Wijs, J.C. van de Pol, E. Bortnik, Solving scheduling problems by untimed model checking – the clinical chemical analyser case study, 11J. Softw. Tools Technol. Transfer (5) (2009) 375–392.
[70] R.A. Zann, The Zebra Finch – A Synthesis of Field and Laboratory Studies, Oxford University Press Inc., 1996.
[71] W. Zhang, State-Space Search – Algorithms Complexity Extensions and Applications, Springer-Verlag, 1999.
[72] R. Zhou, E.A. Hansen, Beam-stack search: integrating backtracking with beam search, Proc. ICAPS 2005, AAAI, 2005, pp. 90–98.