Pruning State Spaces with Extended Beam Search

M. Torabi Dashti and A. J. Wijs

CWI, Amsterdam
{dashti, wijs}@cwi.nl

Abstract. This paper focuses on using beam search, a heuristic search algorithm, for pruning state spaces while generating. The original beam search is adapted to the state space generation setting and two new search variants are devised. The resulting framework encompasses some known algorithms, such as A^*. We also report on two case studies based on an implementation of beam search in μCRL.

1 Introduction

State space explosion is still a major problem in the area of model checking. Over the years a number of techniques have emerged to prune, while generating, parts of the state space that are not relevant given the task at hand. Some of these techniques, such as partial order reduction algorithms (e.g. see [8]), guarantee that no essential information is lost after pruning. Alternatively, this paper focuses mainly on heuristic pruning methods which heavily reduce the generation time and memory consumption but generate only an approximate (partial) state space. The idea is that a user-supplied heuristic function guides the generation such that ideally only relevant parts of the state space are actually explored. This is, in fact, at odds with the core idea of model checking when studying qualitative properties of systems, i.e. to exhaustively search the complete state space to find any corner case bug. However, heuristic pruning techniques can very well target performance analysis problems as approximate answers are usually sufficient.

In this paper, we investigate how *beam search* can be integrated into the state space generation setting. Beam search (BS) is a heuristic method for combinatorial optimisation problems, which has extensively been studied in artificial intelligence and operations research, e.g. see [9, 12]. BS is similar to breadth-first search as it progresses level by level through a highly structured search tree containing all possible solutions to a problem, but it 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 heavily decreases the generation time, but may in general miss essential parts of the search tree, since wrong decisions may be made while pruning. Therefore, BS has so far been mainly used in searching trees with a high density of goal nodes. Scheduling problems, for instance, have been perfect targets for using BS as their goal is to optimally schedule a certain number of jobs and resources, while near-optimal schedules, which densely populate the tree, are in practice good enough.

The idea of using BS in state space generation is an attempt towards integrating functional analysis, to which state spaces are usually subjected, and quantitative analysis. Since model checkers, such as SPIN, UPPAAL and μCRL, which generate these

state spaces, usually have highly expressive input languages, BS for state spaces can be applied in a more general framework compared to its traditional use. Applying BS to search state spaces tightly relates to directed model checking (DMC) [3] and guided model checking (for timed automata) [1], where heuristics are used to guide the search for finding counter-examples to a functional property (usually in LTL) with a minimal exploration of the state space. Using A^* [3] and genetic algorithms [5] to guide the search are among notable works in this field. In contrast to DMC, we generate a partial state space in which an arbitrary property can be checked afterwards (the result would not be exact, hence being useful when near-optimal solutions suffice). However, there are strong similarities as well: A^* can be seen as an instantiation of BS (see § 3.4).

Contributions We motivate and thoroughly discuss adapting the BS techniques to deal with arbitrary structures of state spaces. Next, we extend the classic BS 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. Combinations of these variants create a spectrum of search algorithms that, as will be described, encompasses some known search techniques, such as A^*. We have implemented these variants of BS in the μCRL state space generation toolset [2]. Experimental results for two scheduling case studies are reported.

Road map BS is described in § 2. § 3 deals with the adaptation of existing BS variants to the state space generation setting. There we also propose our extensions to BS. Memory management and choosing heuristics are also discussed there. § 4 reports on two case studies, § 5 presents our related work and § 6 concludes the paper.

2 Beam search

Beam search [9, 12] is a heuristic search algorithm for combinatorial optimisation problems, which has extensively been applied to scheduling problems, for example in systems designed for job shop environments [12].

BS is similar to breadth-first search as it progresses level by level. At each level of the search *tree* (in § 3 we extend BS to handle cycles), it uses a heuristic evaluation function to estimate the promise of encountered nodes [1], while the *goal* is to find a path from the root of the tree to a leaf node that possesses the minimal evaluation value among all the leafs. In 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 to a value before searching starts. Because of this aggressive pruning the search time is a linear function of β and is thus heavily decreased. However, since wrong decisions may be made while pruning, BS is neither complete, i.e. is not guaranteed to find a solution when there is one, nor optimal, i.e. does not guarantee finding an optimal solution. To limit the possibility of wrong decisions, β can be increased at the cost of increasing the required computational effort and memory.

[1] In this section we use nodes and edges, as opposed to states and transitions, to distinguish between the traditional setting of BS and our adaptations.

Two types of evaluation functions have traditionally been used for BS [12]: *priority* evaluation functions and *total-cost* evaluation functions, which lead to the *priority* and *detailed* BS variants, respectively. In priority BS at each node the evaluation function calculates a priority for each successor node and 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. In detailed BS at each node the evaluation function calculates an estimate of the total-cost of the best path that can be found continuing from the current node. 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. When $\beta \to \infty$, detailed and priority BS behave as exhaustive breadth-first search.

3 Adapting beam search for state space generation

Motivation In its traditional setting, BS is typically applied on highly structured search trees, which contain all possible orderings of a given number of jobs, e.g. see [7, 14]. Such a search tree starts with n jobs to be scheduled, which means that the root of the tree has n outgoing transitions. Each node has exactly $n-k$ outgoing transitions, where k is the level in the tree where the node appears. State spaces, however, supposedly contain information on all possible behaviours of a system. Therefore, they may contain cycles, confluence of traces, and 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 of state spaces. Moreover, the BS algorithms search for a particular node (or schedule) in the search space, while in (and after) generating state spaces one might desire to study a property beyond simple reachability. We therefore extend BS to a state space *generation* (SSG) setting, as opposed to its traditional setting that focuses only on *searching*. The notion of a particular "goal" (cf. § 2) is thus removed from the adapted BS (see § 3.5 for possible optimisations when restricting BS to verify reachability properties). This along with the necessary machinery for handling cycles raise memory management issues in BS, that we discuss in § 3.5.

Below, DBS and PBS correspond, respectively, to the detailed and priority beam searches extended to deal with arbitrary state spaces (§ 3.1 and § 3.2). The F and S prefixes refer to the flexible and synchronised variants (§ 3.3 and § 3.4). we start with introducing labelled transition systems.

Labelled transition system (LTS) is a tuple (Σ, s_0, Act, Tr), where Σ is a set of states, $s_0 \in \Sigma$ is the initial state, Act is a finite set of action labels and $Tr \subseteq \Sigma \times Act \times \Sigma$. We write $s \xrightarrow{a} s'$ when $(s,a,s') \in Tr$. In this paper we consider LTSs with finite Σ.

3.1 Priority beam search for state space generation

Below we first present the PBS algorithm and then describe and motivate the changes that we have made to the traditional priority BS.

PBS is shown in figure 1. The sets *Current*, *Next* and *Expanded* denote, respectively, the set of states of the current level, the next level and the set of states that have been expanded. The user-supplied function $priority : Act \to \mathbb{Z}$ provides the priority of

actions, as opposed to states. [2] We motivate this deviation by noting that *jobs* in the BS terminology correspond more naturally with *actions* in LTSs.

The set *Buffer* temporarily keeps seemingly promising transitions. The function $prio_{\min} : \mathscr{P}(Tr) \to \mathbb{Z}$ returns the lowest priority of the actions of a given set of transitions, with $prio_{\min}(\emptyset) = -\infty$. The function $getprio_{\min} : \mathscr{P}(Tr) \to Tr$, given a set of transitions, returns one of the transitions having an action with the lowest priority. Expanding the set $Current \setminus Expanded$ in the **while** loop ensures that no state is revisited in case cycles or confluent traces exist in the search space. The algorithm terminates when it has explored all the states in its beam.

Fig. 1. Priority BS

```
Current := {s0}
Expanded := ∅; Buffer := ∅
level := 0; limit := α
while Current \ Expanded ≠ ∅ do
  Next := ∅
  for all s ∈ Current \ Expanded do
    for all s -a-> s' ∈ en(s) do
      if priority(a) > prio_min(Buffer) then
        if |Buffer| = limit then
          Buffer := Buffer \
                    {getprio_min(Buffer)}
        Buffer := Buffer ∪ {s -a-> s'}
    Next := Next ∪ nxt(s, Buffer)
    Buffer := ∅
  Expanded := Expanded ∪ Current
  Current := Next
  level := level + 1
  if level = l then limit := 1
```

In priority BS, originally, up to β children of the root are selected. The resulting beam of width β is then maintained by expanding only one child per node in subsequent levels. In state spaces, however, the root has typically much less outgoing transitions than the average branching factor of the state space. Fixing the beam width at such an early stage is therefore not reasonable.

To mitigate this problem, instead of β, the algorithm of figure 1 is provided with the pair (α, l), where $\alpha, l \in \mathbb{N}$ and $\alpha^l = \beta$. The idea is that the algorithm uses the *priority* function to prune non-promising states from the very first level, but in two phases: before reaching nearly β states in a single level, it considers the most promising α transitions for further expansion, but after that, it expands only one child per state.

3.2 Detailed beam search for state space generation

In detailed BS a total-cost evaluation function $f : \Sigma \to \mathbb{N}$ is used to guide the search. This function is decomposed into $f(s) = g(s) + h(s)$. The $g(s)$ function represents the cost taken to reach s from the initial state s_0, which is defined as $g(s) = g(s') + cost(a)$ if $s' \xrightarrow{a} s$. The user-supplied function $cost : Act \to \mathbb{N}$ assigns weights to actions that can, e.g., denote the time needed to perform different jobs in a scheduling problem. These weights are fixed before search starts. Since the range of *cost* is non-negative numbers, we have $s \to^* s' \implies g(s') \geq g(s)$. The user-supplied function $h(s)$ estimates the cost it would take to efficiently reach a goal state (or complete the schedule) continuing from s.

[2] In general, *priority* can also depend on states: $priority : \Sigma \to Act \to \mathbb{Z}$. In this paper, we only consider fixed priorities, which resembles *dispatch* scheduling in AI terminology [12].

Thus, for a goal state s, $h(s) = 0$. The f function is called *monotonic* if $s \rightarrow^* s'$ implies $f(s) \leq f(s')$.

The original idea of detailed BS does not need to change much to fit into the SSG setting except for when handling cycles. When exploring a cyclic LTS, to guarantee the termination of the algorithm, it is necessary to store the set of explored states to avoid exploring a state more than once (cf. the *Expanded* set in figure 1). However, if a state is reached via a path with a lower cost, the state has to be re-examined. This is because the total-cost of each state depends on the cost to reach that state from the root, cf. § 3.4.

3.3 Flexible beam search

A major issue that still remains unaddressed in the BS adaptations of § 3.1 and 3.2 is the *tie-breaking* problem: How should equally competent candidates, e.g. having the same f values, be pruned? These selections are beyond the influence of the evaluation function and can undesirably make the algorithm non-deterministic. Hence, we propose two variants of BS that we call *flexible detailed* and *flexible priority* beam searches, in which the beam width can change during state space generation.

In flexible detailed BS, 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 priority BS, at each state, all the transitions with the same priority as the most promising transition of that particular state are selected. Note that in FPBS, in contrast to FDBS, if the beam width is stretched, it cannot be readjusted to the intended β.

3.4 Synchronised beam search

As is described in § 2, the classic BS algorithms were tailored for the breadth-first exploration strategy. Below, we explain a way to do BS on top of best-first [10] exploration algorithms. Broadly speaking, we separate the exploration strategy from the pruning phase, so that the exploration is guided with a (possibly different [3]) heuristic function. This is particularly useful when checking reachability properties on-the-fly.

Below, we inductively describe $\mathscr{G}$*-synchronised* xBS, where $\mathscr{G} : \Sigma \rightarrow \mathbb{N}$ is the function that guides the exploration and $x \in \{\mathrm{D}, \mathrm{P}, \mathrm{FD}, \mathrm{FP}\}$ (denoting the BS variants described previously). Let $\hat{S}_i$ denote the set of states to be explored at round i. [4] We partition this set into equivalence classes $c_0, \cdots, c_n$, where $n \in \mathbb{N}$, such that $\hat{S}_i = c_0 \cup \cdots \cup c_n$ and $\forall s \in \hat{S}_i.\ s \in c_j \iff \mathscr{G}(s) = j$. The pruning algorithm xBS is subsequently applied only on c_k where $c_k \neq \emptyset \ \wedge\ \forall j < k.\ c_j = \emptyset$. According to the pruning algorithm (which can possibly employ an evaluation function different from $\mathscr{G}$), some of the successors of c_k are selected, constituting the set $\hat{S}$. The next round starts with $\hat{S}_{i+1} = \hat{S} \cup \hat{S}_i \setminus c_k$. Since synchronised beam search separates the exploration algorithm from the pruning algorithm, it can be perfectly combined with the other variants of BS introduced earlier.

[3] Using different functions for guiding exploration and pruning in principle allows dealing with multi-priced optimisation problems, cf. [1].

[4] "Round" i corresponds to a logical (i.e. not necessarily horizontal) level in the state space, which is processed in the i^{th} iteration of SSG algorithm.

Using any constant function as $\mathscr{G}$ in SDBS would clearly result in BS with breadth-first exploration strategy.

Figure 2 shows $\mathscr{G}$-SDBS in detail. The sets *Current*, *Next* and *Expanded* contain pairs of states and corresponding g values, i.e. $\langle s, s.g\rangle$. The function $get\, f_{\max} : \mathscr{P}(\Sigma) \to \Sigma$, given a set of states, returns one of the states that has the highest f value. Here $unify(X)$ and $update(X,Y)$ are defined as follows: $unify(X) = \{\langle s,g\rangle \in X \mid \forall \langle s,g'\rangle \in X.\ g \leq g'\}$ and $update(X,Y) = \{\langle s,g\rangle \in X \mid \neg\exists \langle s,g'\rangle \in Y.\ g' \leq g\}$. In this algorithm, a state will be revisited only if it is reached via a path with a lower cost than the g cost assigned to it (see also § 3.2).

Fig. 2. Synchronised detailed BS

```
s0.g := 0; Current := {⟨s0, s0.g⟩}
Expanded := ∅
while Current ≠ ∅ do
  Next := ∅; i := −1; found := F
  while found = F do
    i := i + 1
    ci := {⟨s, s.g⟩ ∈ Current | G(s) = i}
    if ci ≠ ∅ then
      Current := Current \ ci
      found := T
  while |ci| > β do
    ci := ci \ {get fmax(ci)}
  for all s ∈ ci do
    for all s −a→ s' ∈ en(s) do
      s'.g := s.g + cost(a)
      Next := Next ∪ {⟨s', s'.g⟩}
  Expanded := unify(Expanded ∪ ci)
  Current := update(unify(Next ∪
                 Current), Expanded)
```

To mention a practically interesting candidate for $\mathscr{G}$, we temporarily deviate from our general setting. Consider the problem of finding a path of minimal cost that leads to a particular state in the search space. 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 the root to s. If $\mathscr{G}(s) = g(s)$ in $\mathscr{G}$-synchronised DBS, once the goal state is found, searching can safely terminate. This is because at 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 $g(s) \leq g(s')$. We observe that in g-SDBS no state is re-explored, because states with minimal g are taken first and thus a state can be reached again only via paths with higher costs (cf. § 3.2). Both g-SDBS and g-SPBS have been used in our experiments of § 4, where minimal-time traces to a particular state are searched for.

As another variant of synchronised search, we note that given a *monotonic* total-cost function $f(s) = g(s) + h(s)$ (cf. § 3.2), f-SFDBS with arbitrary $\beta > 0$, corresponds to the well-known A^* search algorithm (e.g. see [10]). [5] Due to space constraints, we refer to [13] for a proof of this relation.

3.5 Discussions

Memory management is a challenging issue in SSG. Although BS reduces memory usage due to cutting away parts of the state space, still explored states need to be accessed to guarantee the termination of SSG in case of cyclic LTSs. This can be partially

[5] The monotonicity assumption on f is necessary for optimality of A^* [10].

counter-measured by taking into account specific characteristics of the problem at hand and the properties that are to be checked:
1. When aiming at a reachability property (such as reachability of a goal state, checking invariants and hunting deadlock states), once a state satisfying the desired property is reached the search can terminate and the witness trace can be reported. This however cannot be extended to arbitrary properties.
2. If there are no cycles in the state space, 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 [6], i.e. flushed to high latency media such as disks.

Heuristic functions and selecting the beam width Effectiveness of BS hinges on selecting good heuristic functions. Heuristic functions heavily depend on the problem being solved. As our focus here is on exploration strategies that utilise 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 of them. Among others, [3, 6, 12] complement the work we present in this paper, as they explain how to design heuristic functions when, e.g., analysing Java programs or provide approximate distance to deadlocks, etc.

Selecting the beam width β is another challenge in using BS. The beam width intuitively calibrates the time and memory usage of the algorithm on one hand and the accuracy of the results on the other hand. Therefore, in practice the time and memory limits of a particular experiment determine β. To reduce the sensitivity of the results to the exact value of β, flexible BS variants can be used. This, however, comes at the price of losing a tight grip on the memory consumption (see also § 3.3). For a general discussion on selecting β and its relation to the quality of answer see [12].

4 Experimental results

In this section we report our experimental results. [7]

Cannibals and missionaries (C&M) problem is a classic river crossing puzzle and produces state spaces with interesting structures: they contain cycles, deadlocks and confluent traces. Assume that *C* missionaries and *C* cannibals stand on the left bank of a river that they wish to cross. The goal is to find a schedule for ferrying all the cannibals and all the missionaries across using a boat that can take up to *B* people at a time. The condition is that the cannibals never outnumber the missionaries, on a shore or in the boat. Moving the boat costs 1 time unit per passenger, and we wish to find a minimal cost path towards the goal. We use a μCRL implementation of BS and a SPIN implementation of the depth-first branch-and-bound algorithm to solve this problem. The

[6] In this case, some states may be revisited due to confluent traces, hence undesirably increasing the search time.

[7] 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. See `http://www.cwi.nl/~wijs/TIPSy` for a complete report along with specs.

results are shown in table 1. Since μCRL and SPIN count states in different ways, the numbers of states of the experiments using different tools are not comparable.

In μCRL, we first applied g-SFDBS with constant h (i.e. no estimation) with any $\beta > 0$, denoted minimal cost search MCS in table 1. MCS is an exhaustive search method, where the states are ordered based on the cost needed to reach them from the initial state. This search is used to find the minimum number of time units needed to solve the problem (shown in the *Result* column). As a comparison, we have also performed experiments with SPIN. In those cases, we followed the algorithm of [11], a prominent technique to use heuristics within SPIN. The idea is that the LTL formula that is checked is modified during verification to reflect the best solution found so far. This can effectively implement a branch-and-bound mechanism in SPIN, denoted DFS BnB Prop in table 1. This algorithm avoids exhaustive search, yet it is complete.

Besides that we used g-SFDBS with $h(s) = C(s) + M(s) + (\langle C(s) \neq M(s) \rangle \times (2 \times C))$ as the heuristic part of DBS, where $C(s)$ and $M(s)$ are the numbers of cannibals and missionaries on the left bank in state s, respectively, and $\langle C(s) \neq M(s) \rangle$ is a Boolean expression returning 1 if $C(s) \neq M(s)$, and 0 otherwise. In table 1, the 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. Our g-SFDBS algorithm should ideally be compared with other heuristic state space generation tools, such as HSF-SPIN [4]. We however leave this as future work.

Table 1. Experimental results C&M. Times are in min:sec. o.o.t.: out of time (set to 12 hours); o.o.m.: out of memory (set to 900 MB)

Problem	Result	μCRL MCS		μCRL g-SFDBS				SPIN DFS		SPIN DFS BnB Prop.	
(C,B)	T	# States	Time	T	β	# States	Time	# States	Time	# States	Time
(3,2)	18	147	00:03.80	18	3	142	00:03.73	28,535	00:00.32	26,062	00:00.29
(20,4)	104	2,537	00:05.32	106	10	2,191	00:05.38	445,801	00:02.66	408,053	00:02.34
(50,20)	116	90,355	00:20.15	120	15	17,361	00:11.45	12,647,000	02:05.25	12,060,300	01:49.59
(100,10)	292	49,141	00:19.65	296	10	16,274	00:14.46	14,709,600	02:49.32	13,849,300	02:23.34
(100,30)	222	366,608	01:05.79	228	15	61,380	00:32.06	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(300,30)	680	1,008,436	04:10.72	684	15	205,556	02:30.11	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(500,50)	1,076	4,365,536	21:40.52	1,080	20	685,293	10:33.28	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(500,100)	1,036	17,248,979	77:16.36	1,040	20	1,170,242	16:47.10	o.o.m.	o.o.m.	o.o.m.	o.o.m.
(1000,250)	o.o.t.	o.o.t.	o.o.t.	2,032	20	5,317,561	240:22.11	o.o.m.	o.o.m.	o.o.m.	o.o.m.

Clinical Chemical Analyser (CCA) is a case study taken from industry [15]: it is used to analyse batches of test receipts on patient samples (blood, plasma, etc) that are uniquely described by a triple which indicates the number of samples of each fluid (see table 2). We have extensively described the CCA case in [16].

Table 2 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. [8]

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 CCA provides a case study which can better be tackled using priority BS, compared to detailed BS variants.

Table 2. Experimental results CCA. o.o.t.: out of time (set to 30 hours)

Case	Result	MCS		g-SDBS			g-SPBS			g-SFPBS	
		β	Time	β	#States	Time	(α, l)	#States	Time	#States	Time
(3,1,1)	36	3,375	00:10.35	25	1,461	00:03.43	1,1	48	00:03.03	821	00:03.70
(1,3,1)	39	13,194	00:30.48	41	2,234	00:03.93	1,1	179	00:03.08	1,133	00:04.06
(6,2,2)	51	341,704,322	1524:56.00	81	7,408	00:07.76	2,9	479	00:03.06	45,402	02:33.65
(1,2,7)	73	o.o.t.	o.o.t.	75,000	6,708,705	84:38.41	1,1	90	00:02.99	122,449	04:02.94
(7,4,4)	75	o.o.t.	o.o.t.	35,000	3,801,607	41:01.80	3,25	155,379	08:14.66	20,666,509	872:55.71

5 Related work

BS is extended to a complete search in [18], by using a new data structure, called a beam stack. Thereby, it is possible to achieve a range of searches, from depth-first search ($\beta = 1$) to breadth-first search ($\beta \to \infty$). Considering our extensions for arbitrary state spaces, it would be interesting to try to combine these two approaches.

Notable works on scheduling using formal method tools are [1] and [11]. In [1], Behrmann et al. have extended timed automata with linearly priced transitions and locations, resulting in UPPAAL CORA tool. They deal with reachability analysis using the standard branch-and-bound algorithm. A number of basic exploration techniques can be used for branching, and bounding is done based on heuristics. In [11], the depth-first branch-and-bound technique is used for scheduling in SPIN. See also § 4.

In [17], we report on a distributed implementation of the BS variants proposed in this paper, where a number of machines together perform these search algorithms.

6 Conclusions

In this paper, we extended and made available an existing search technique to be used for quantitative analysis within a setting used for system verification.

[8] In FPBS once the beam width is stretched, it cannot be readjusted to its initial value, see § 3.3.

Our experiments showed the usefulness and flexibility of these extensions. We observed that BS can be tuned to encompass some other (heuristic) search algorithms, thus providing a flexible state space generation framework.

Future work Comparing our implementation with other heuristic state space generation tools, such as HSF-SPIN, is certainly a next step for this work. Also, BS can in principle deal with infinite state spaces given that the heuristic function does not cut away all finite paths. This application of BS has yet to be investigated.

Acknowledgements We are grateful to Jaco van de Pol and Michael Weber for their insightful comments on the paper, and to Bert Lisser for implementing parts of BS variants.

References

1. G. Behrmann, K. Larsen, and J. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.
2. S. Blom, W. Fokkink, J. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μCRL: A toolset for analysing algebraic specifications. In *CAV'01*, volume 2102 of *LNCS*, pages 250–254, 2001.
3. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *STTT*, 5(2):247–267, 2004.
4. S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *SPIN '01*, pages 57–79. Springer, 2001.
5. P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *TACAS'02*, volume 2280 of *LNCS*, pages 266–280. Springer, 2002.
6. A. Groce and W. Visser. Heuristics for model checking Java programs. *STTT*, 6(4):260–276, 2004.
7. S. Oechsner and O. Rose. Scheduling cluster tools using filtered beam search and recipe comparison. In *Proc. 2005 Winter Simulation Conference*, pages 2203–2210. IEEE, 2005.
8. D. Peled. Ten years of partial order reduction. In *CAV '98*, volume 1427 of *LNCS*, pages 17–28. Springer, 1998.
9. M. Pinedo. *Scheduling: Theory, algorithms, and systems*. Prentice-Hall, 1995.
10. S. Russell and P. Norvig. *Artificial intelligence: A modern approach*. Prentice-Hall, 1995.
11. T. Ruys. Optimal scheduling using Branch-and-Bound with SPIN 4.0. In *SPIN'03*, volume 2648 of *LNCS*, pages 1–17, 2003.
12. P. Si Ow and E. Morton. Filtered beam search in scheduling. *Intl. J. Production Res.*, 26:35–62, 1988.
13. M. Torabi Dashti and A.J. Wijs. Pruning state spaces with extended beam search. Technical Report SEN-R0610, CWI, 2006. `ftp.cwi.nl/CWIreports/SEN/SEN-R0610.pdf`.
14. J. Valente and R. Alves. Filtered and recovering beam search algorithms for the early/tardy scheduling problem with no idle time. *Comput. Ind. Eng.*, 48(2):363–375, 2005.
15. S. Weber. *Design of Real-Time supervisory control systems*. PhD thesis, TU/e, 2003.
16. A. Wijs, J. van de Pol, and E. Bortnik. Solving scheduling problems by untimed model checking. In *Proc. FMICS'05*, pages 54–61. ACM Press, 2005.
17. A.J. Wijs and B. Lisser. Distributed extended beam search for quantitative model checking. In *MoChArt'06*, volume 4428 of *LNAI*, pages 165–182, 2007.
18. R. Zhou and E. Hansen. Beam-stack search: Integrating backtracking with beam search. In *Proc. ICAPS'05*, pages 90–98. AAAI, 2005.