Beyond Red-Black Planning: Limited-Memory State Variables
(technical report)

Patrick Speicher  Marcel Steinmetz  Daniel Gnäd Jörg Hoffmann
Saarland Informatics Campus, Saarland University
Saarbrücken, Germany
speicher@uni-saarland.de; {steinmetz, gnad, hoffmann}@cs.uni-saarland.de

Alfonso Gerevini
University of Brescia
Brescia, Italy
gerevini@ing.unibs.it

Abstract
Red-black planning delete-relaxes only some of the state variables. This is coarse-grained in that, for each variable, it either remembers all past values (red), or remembers only the most recent one (black). We herein introduce limited-memory state variables, that remember a subset of their most recent values. It turns out that planning is still PSPACE-complete even when the memory is large enough to store all but a single value. Nevertheless, limited memory can be used to substantially broaden a known tractable fragment of red-black planning, yielding better heuristic functions in some domains.

Introduction
The delete relaxation has been instrumental for scalability in satisficing classical planning (e.g., (Bonet and Geffner 2001; Hoffmann and Nebel 2001; Gerevini, Saetti, and Serena 2003; Richter and Westphal 2010)), but it also has weaknesses. Partial delete relaxation methods interpolate between delete-relaxed planning and real planning (Keyder, Hoffmann, and Haslum 2012; 2014; Katz, Hoffmann, and Domshlak 2013; Domshlak, Hoffmann, and Katz 2015). We herein focus on red-black planning, which applies the delete-relaxed semantics to a subset of state variables (the "red" ones), letting them accumulate their values, while keeping the real semantics for the others (the "black" ones). Red-black planning is tractable if the dependencies between black variables are acyclic, and each black variable is invertible. The heuristic function based on that tractable fragment, $h^\text{RB}$, can be quite useful; it was a key part of the Mercury system that performed well in IPC’14.

Yet distinctions at the level of entire state variables are very coarse-grained: either we remember all past values of a variable (red), or only the most recent one (black). We herein introduce limited-memory state variables, that allow more fine-grained relaxations through remembering a subset of their most recent values. As we show, planning is still PSPACE-complete even when the memory is large enough to store all but a single value. Nevertheless, limited memory can be used to substantially extend the abovementioned tractable fragment. In $h^\text{RB}$, non-invertible $v$ cannot be painted black, because $v$ may not be able to go back to a previous value when required. Our idea is, instead of fully delete-relax such $v$, to give $v$ "just enough" memory to ensure this property. The resulting heuristic function proves to be superior in some domains.

Preliminaries
We use the finite-domain representation (FDR) framework. We introduce FDR and its delete relaxation as special cases of red-black planning. A red-black (RB) planning task is a tuple $I = \langle V^B, V^R, A, I, G \rangle$. $V^B$ is a set of black variables and $V^R$ is a set of red variables, where $V^B \cap V^R = \emptyset$ and each $v \in V := V^B \cup V^R$ has a finite domain $D[v]$. The initial state $I$ is a complete assignment to $V$, the goal $G$ is a partial assignment. Each action $a$ is a pair $\langle \text{pre}_a, \text{eff}_a \rangle$ of partial assignments called precondition and effect. For a partial assignment $p, V(p)$ denotes the subset of $V$ instantiated by $p$. For $V' \subseteq V(p), p[V']$ denotes the value of $V'$ in $p$.

A state $s$ assigns each $v \in V$ a non-empty subset $s[v] \subseteq D[v]$, where $|s[v]| = 1$ for all $v \in V^B$. An action $a$ is applicable in $s$ if $\text{pre}_a[v] \subseteq s[v]$ for all $v \in V(\text{pre}_a)$. Applying $a$ in $s$ changes the value of $v \in V(\text{eff}_a) \cap V^R$ to $\text{eff}_a[v]$, and changes the value of $v \in V(\text{eff}_a) \cap V^R$ to $s[v] \cup \{\text{eff}_a[v]\}$. The resulting state is denoted $s[a]$. A plan is an action sequence whose iterative application in $I$ leads to a state $s$ where $G[v] \in s[v]$ for all $v \in V(G)$.

If $I$ is an FDR planning task if $V = V^B$, and is a monotonic (MFDR) planning task if $V = V^R$. The delete relaxation uses MFDR to approximate FDR; red-black relaxation uses the more general RB instead. The state-of-the-art red-black heuristic, $h^\text{RB}$, does so via choosing a variable painting – a partitioning of $V$ into $V^B$ and $V^R$ – such that the causal graph over $V^B$ is acyclic, and each $v \in V^B$ is relaxed side effects invertible. We will explain these notions later on when considering our extended heuristic function.

Limited-Memory State Variables
Our notion of limited-memory variables is an instance of what we baptize trace-memory relaxation. In its most general form, such a relaxation $\mathcal{R}$ is defined through a function $\mathcal{R}[v] : D[v]^+ \rightarrow 2^{D[v]}$ for each variable $v$. For a given value history $\delta \in D[v]^+$ ($v$'s value sequence on the current plan prefix), $\mathcal{R}[v](\delta)$ is the value subset that $v$ will "remember" under $\mathcal{R}$. We apply the following two restrictions to $\mathcal{R}[v]$: (1) For all $v \in V$ and $\delta = d_1, \ldots, d_n \in D[v]^+$, $\mathcal{R}[v](\delta) \subseteq \{d_1, \ldots, d_n\}$. (2) the "latest" value $d_n$ is in $\mathcal{R}[v](\delta)$. Given the state history $I = s_0, a_1, \ldots, a_{m-1}, s_m = s$, we de-
The delete relaxation uses $R[v](d_1, \ldots, d_n) := \{d_1, \ldots, d_n\}$. Red-black relaxation uses the same for $v \in V^R$, and uses $R[v](d_1, \ldots, d_n) := \{d_n\}$ for $v \in V^B$. Our new limited-memory relaxation is parameterized by a memory size $M[v]$, $1 \leq M[v] \leq |D[v]|$, for every $v \in V$. It uses $R'[v](d_1, \ldots, d_n) := \{d_k, \ldots, d_n\}$, where $k = 1$ if $|\{d_1, \ldots, d_n\}| \leq M[v]$ and otherwise $k$ is s.t. $|\{d_k, \ldots, d_n\}| = M[v]$. In words, each variable remembers as many of its recent values as fit into memory. Observe that $v \in V^R$ is characterized by $M[v] = |D[v]|$, and $v \in V^B$ is characterized by $M[v] = 1$.

The space of trace-memory relaxations is a refinement hierarchy, where $R$ refines $R'$ if, for every $v \in V$ and $\delta \in D[v]^+$, we have $R'[v](\delta) \subseteq R'[v](\delta)$. If $R$ refines $R'$, then $R$ is more informed than $R'$. Any $R$-relaxed plan is an $R'$-relaxed plan, but not vice versa (unless $R = R'$). The unique coarsest element in the refinement hierarchy is the delete relaxation, $R^+$, as every $R$ refines $R^+$. The unique most refined element is the standard FDR semantics, $R^*$, as $R^*$ refines every $R'$. A limited-memory relaxation $R$ refines a red-black relaxation iff $M[v] = 1$ for all $v \in V^B$.

Worst-Case Complexity

We consider the complexity of $R$-relaxed plan existence for limited-memory relaxations $R$. Observe first that $R$-relaxed plan existence remains a member of PSPACE, thanks to a similar non-deterministic polynomial space algorithm as given for $R^*$ by Bylander (1994) (it suffices to remember the most recent $M[v]$ values for each $v$). Observe further that, likewise easily, $R$-relaxed plan existence in full generality remains PSPACE-hard, simply as it includes the case where $R = R^*$ and we do not actually relax anything. That case is, of course, not interesting, so we will exclude it. Observe finally that limited-memory relaxation is pointless for binary variables $v$, i.e., for $|D[v]| = 2$, as one can only either set $M[v] = 2$ (delete relaxation), or $M[v] = 1$ (no relaxation).

We hence focus on the case where, for all $v$, $|D[v]| > 2$ and $M[v] > 1$. Somewhat surprisingly perhaps, it turns out that this remains PSPACE-hard even in the maximally relaxed setting, where $M[v] \geq |D[v]| - 1$ for all $v$, i.e., at most a single value per variable is forgotten:

**Theorem 1** For FDR tasks where $|D[v]| > 2$ for all $v$, and limited-memory relaxations $R$ where $M[v] \geq |D[v]| - 1$ for all $v$, deciding $R$-relaxed plan existence is PSPACE-hard.

**Proof Sketch:** Our proof is by reduction from FDR with binary variables, $D[v] = \{0, 1\}$ for all $v$, and where $\text{pred}_a[v] = 0$ whenever $\text{eff}_a[v] = 1$, as well as $\text{pred}_a[v] = 1$ whenever $\text{eff}_a[v] = 0$. Bylander’s (1994) PSPACE-hardness proof (his Theorem 3.1) is easy to adapt to this FDR fragment.

Our key observation is that any binary variable $v$ can be encoded into a counter, consisting of three counter variables $v_1, v_2, v_3$ with domain $D[v_i] = \{1, 2, 3, 4\}$ and $M[v_i] = 3$. The counter is arranged to behave as follows:

![Counter](image)

The counter has a life cycle through 4 counter states – memory contents of the counter variables – of which state 1 encodes $v = 0$ and state 4 encodes $v = 1$. State 0 is characterized by the single value $i$ all counter variables have in common (left-top to right-bottom diagonal in the figure). To achieve this behavior, the action moving the counter from state $j$ to state $j + 1$ has precondition $\{v_1 = j, v_2 = j, v_3 = j\}$, and adds a new value to each $v_i$ corresponding to the new rightmost column in the figure. Any precondition/goal $v = 0$ can then be replaced by $\{v_1 = 1, v_2 = 1, v_3 = 1\}$, and any precondition/goal $v = 1$ can be replaced by $\{v_1 = 4, v_2 = 4, v_3 = 4\}$. To move from counter state 1 ($v = 0$) to counter state 4 ($v = 1$), we insert two dummy actions which must be applied, and will be applicable only, after an effect $v := 1$ (which by construction necessitates a precondition $v = 0$ so will happen only in counter state 1).

The construction assumes that, at the start of the lifecycle, the counter is in state 1, which cannot be specified in FDR as each $v_i$ has only one initial value. But that issue can be solved through an additional counter-initialization phase.

A full proof of this theorem is available in the appendix.

Extending $h^{\text{RB}}$ with Non-Invertible Variables

Towards explaining our extension of $h^{\text{RB}}$, let us summarize the workings of that heuristic function. To evaluate a state $s$, $h^{\text{RB}}(s)$ first computes a fully delete relaxed plan $\pi^+$, and then refines $\pi^+$ to treat the variables $V^B$ accordingly. The refinement process goes through $\pi^+ = (a_1^+, \ldots, a_k^+)$ from front to back, executes each $a_i^+$ using the RB semantics, and inserts a repair sequence $\pi^B$ for $v \in V^B$ whenever the precondition of $a_{i+1}^+$ on $v$ is not satisfied.¹ That sequence $\pi^B$ is found by solving a planning task $\Pi^B$ over the black variables $V^B$. The initial state of $\Pi^B$ is the state of $V^B$ before executing $a_{i+1}^+$, the goal is the precondition of $a_{i+1}^+$. We denote these by $I^B$ and $G^B$ respectively.

For $h^{\text{RB}}$ to be tractable, solving $\Pi^B$ must be tractable. To that end, $h^{\text{RB}}$ relies on a known tractability result for planning, restricting (a) cross-variable dependencies to be acyclic, and restricting (b) each variable to be able to move from any value $d$ to any other value $d'$ (Knoblock 1994; Williams and Nayak 1997; Brafman and Domshlak 2003; Helmert 2006; Chen and Giménez 2010). Figure 1 shows the algorithm, repair planning, that $h^{\text{RB}}$ uses in that setting.

Repair planning sequentializes the handling of black variables, from “clients” to “servants” according to prerequisite (a) above. Formally, (a) is captured in terms of the causal graph over $V^B$ (Knoblock 1994; Brafman and Domshlak 2003; Helmert 2006). In the acyclic case, every action

¹Actually, this is what Domshlak et al. (2015) call relaxed plan repair, a simpler and empirically worse variant of the red facts following algorithm $h^{\text{RB}}$ uses. We explain relaxed plan repair as red facts following is complicated, and our extensions are identical for both algorithms. Our implementation extends red facts following.
We implemented \( h^{\text{Gray}} \) in FD (Helmer 2006), extending Domshlak et al.’s (2015) implementation of \( h^{RB} \). We adopt Domshlak et al.’s \textit{stop search} technique, which tests whether the relaxed plan is actually a real plan, and if so, stops the search. We use this in all configurations tested.

We compare \( h^{\text{Gray}} \) to its direct predecessor \( h^{RB} \), and to \( h^{FF} \) as a baseline. We use FD’s canonical search algorithm for satisficing planning, greedy best-first search (GBFS). To get an unbiased comparison between the different heuristics, respectively to compare to state-of-the-art, we report results from disabling, as well as enabling FD’s preferred operators. For the latter, to enhance comparability, we use the same preferred operators, taken from \( h^{FF} \), for all three heuristics.\(^2\)

\(^2\)Taking preferred operators from the individual heuristics does not significantly affect the results.
To choose the partitioning of $V$ into $V^B$, $V^G$, and $V^R$, we extend Domshlak et al.’s painting strategy. That strategy starts with $V^B = V$. It then paints red, i.e., moves to $V^R$, (1) all causal graph leaf variables, and (2) all non-RSE-invertible variables. From the remaining variables $V^B$, (3) iteratively one $v$ is picked and painted red, until the causal graph over $V^B$ is acyclic. We use the same strategy, except that we omit step (2) and, upon termination, paint all non-RSE-invertible $v \in V^B$ gray (moving them to $V^G$).

To pick $v$ in (3), Domshlak et al. experiment with many prioritization options. Here we use their canonical (simple and performant) option, always selecting the $v$ with highest index in Helmert’s (2004) level ordering. More precisely, we experiment with two different strategies, (A) as just described, (B) preferring non-RSE-invertible $v$ and breaking ties by the level ordering (so that RSE-invertible $v$ are more likely to end up black). With (B), $h^{Gray}$ is very close to $h^{RB}$. With differences only where some variables can be painted gray on top of the black ones in $h^{RB}$. With (A), the paintings tend to differ more. For space reasons, we show data only for (A) and briefly summarize the differences for (B).

We run all IPC STRIPS benchmarks. We assume unit ac-
by the German Research Foundation (DFG), under grant HO 2169/5-1, “Critically Constrained Planning via Partial Delete Relaxation”, and was partially supported by the German Federal Ministry of Education and Research (BMBF) through funding for the Center for IT-Security, Privacy and Accountability (CISPA, grant no. 16KIS0656).

References


Proofs

Theorem 2 For FDR tasks where \(|D[v]| > 2\) for all \(v\) and limited-memory relaxations \(R\) where \(M[v] \geq |D[v]| - 1\) for all \(v\), deciding \(R\)-relaxed plan existence is \(\text{PSPACE}\)-hard.

Proof: Our proof is by reduction from FDR with binary variables, \(D[v] = \{0, 1\}\) for all \(v\), and where \(\text{pre}_a[v] = 0\) whenever \(\text{eff}_a[v] = 1\), as well as \(\text{pre}_a[v] = 1\) whenever \(\text{eff}_a[v] = 0\). Bylander’s (1994) \(\text{PSPACE}\)-hardness proof (his Theorem 3.1) is easy to adapt to this FDR fragment.

Our key observation is that any binary variable \(v\) can be encoded into a counter, consisting of three counter variables \(v_1, v_2, v_3\) with domain \(D[v_i] = \{1, 2, 3, 4\}\) and \(M[v_i] = 3\). The counter is arranged to behave as follows:

\[
\begin{align*}
1: & \quad v = 0 & 2: & \quad \text{dum1} & 3: & \quad \text{dum2} & 4: & \quad v = 1 \\
1 & 2 & 3 & v := 1 & 2 & 3 & 4 & 1 & 2 \\
4 & 1 & 2 & 3 & 4 & 1 & 2 & v := 1 & 2 & 3 & 4 & 1 & 2 \\
4 & 3 & 4 & 1 & 2 & v := 1 & 2 & 3 & 3 & 4 & 1 & 2 & 3 & 4
\end{align*}
\]

The counter has a life cycle through 4 counter states — memory contents of the counter variables \(v\) — of which state 1 encodes \(v = 0\) and state 4 encodes \(v = 1\). State 1 is characterized by the single value \(i\) all counter variables have in common (left-top to right-bottom diagonal in the figure).

To achieve this behavior, the action moving the counter from state 1 to state 2 has precondition \(\{v_1 = j, v_2 = j, v_3 = j\}\), and adds a new value to each \(v_i\) corresponding to the new rightmost column in the figure. Any precondition/goal \(v = 0\) can then be replaced by \(\{v_1 = 1, v_2 = 1, v_3 = 1\}\), and any precondition/goal \(v = 1\) can be replaced by \(\{v_1 = 4, v_2 = 4, v_3 = 4\}\). To move from counter state 1 \((v = 0)\) to counter state 4 \((v = 1)\), we insert two dummy actions which must be applied, and will be applicable only, after an effect \(v := 1\) (which by construction necessitates a precondition \(v = 0\) so will happen only in counter state 1).

In detail, there are four types of actions interacting with the counter: (1) ones that change the value of \(v\) from 0 to 1, corresponding to the counter state change from 1 to 2; (2) ones that change the value of \(v\) from 1 to 0, corresponding to the counter state change from 4 to 1; (3) the dummy action changing the counter state from 2 to 3; (4) the dummy action changing the counter state from 3 to 4. For this setup, we modify the action set as follows:

(1) Any action \(a\) with \(\text{pre}_a[v] = 0\) and \(\text{eff}_a[v] = 1\) is replaced by \(a'\) with \(\text{pre}_a' = \{v_1 = 1, v_2 = 1, v_3 = 1\} \cup \text{pre}_a[V \setminus \{v\}]\) and \(\text{eff}_a' = \{v_1 = 4, v_2 = 3, v_3 = 2\} \cup \text{eff}_a[V \setminus \{v\}]\).

(2) Any action \(a\) with \(\text{pre}_a[v] = 1\) and \(\text{eff}_a[v] = 0\) is replaced by \(a'\) with \(\text{pre}_a' = \{v_1 = 4, v_2 = 4, v_3 = 4\} \cup \text{pre}_a[V \setminus \{v\}]\) and \(\text{eff}_a' = \{v_1 = 3, v_2 = 2, v_3 = 1\} \cup \text{eff}_a[V \setminus \{v\}]\).
(3) We introduce a new dummy action $dum_1$ with $pre_{dum_1} = \{ v_1 = 2, v_2 = 2, v_3 = 2 \}$ and $eff_{dum_1} = \{ v_1 = 1, v_2 = 4, v_3 = 3 \}$.

(4) We introduce a new dummy action $dum_2$ with $pre_{dum_2} = \{ v_1 = 3, v_2 = 3, v_3 = 3 \}$ and $eff_{dum_2} = \{ v_1 = 2, v_2 = 1, v_3 = 4 \}$.

Observe that, beside the respective dummy action, no other action affecting $v_1, v_2, v_3$ is applicable if the counter is in state 2 or 3; and no dummy action is applicable if the counter is in state 1 or 4. Thus, the counter as a whole behaves exactly like $v$, if we abstract away the dummy actions.

The above specification is for encoding a single variable $v$. To encode the overall set of variables, we simply process each the variables one at a time. Observe that, for each variable $v$ in this process, the encoding takes time and space $O(k)$ where $k$ is the number of actions affecting $v$. Hence, in particular, the overall encoding time is polynomial in the size of the input task.

The construction so far assumes that, at the start of the lifecycle, the counter is in state 1. Yet that cannot be specified in FDR as each $v_i$ has only one initial value. But that issue can be solved through an additional counter-initialization phase.

Namely, we set the leftmost column (143) in $I$, and introduce two initializing actions $av_{I}^1$ and $av_{I}^2$ adding the columns 214 and 321 respectively, where $av_{I}^2$ has precondition 214 to enforce the correct initialization order. To enforce a strict separation between the initializing and lifecycle phases, we introduce a guard variable $v^g$ with $D[v^g] = \{1, 2, 3\}$ and $M[v^g] = 2$. We set $I(v^g) = 1$, give $av_{I}^1$ precondition $v^g = 1$ and effect $v^g = 2$, give $av_{I}^2$ precondition $v^g = 1$ and effect $v^g = 3$, and introduce $v^g = 3$ into the precondition of every other action. That is, we modify the action set as follows:

- For any action $a$, we set $pre_a := pre_a \cup \{v^g = 3\}$.
- We introduce a new dummy action $av_{I}^1$ with $pre_{av_{I}^1} = \{ v_1 = 1, v_2 = 4, v_3 = 3, v^g = 1 \}$ and $eff_{av_{I}^1} = \{ v_1 = 2, v_2 = 1, v_3 = 4, v^g = 2 \}$.
- We introduce a new dummy action $av_{I}^2$ with $pre_{av_{I}^2} = \{ v_1 = 2, v_2 = 1, v_3 = 4, v^g = 1 \}$ and $eff_{av_{I}^2} = \{ v_1 = 3, v_2 = 2, v_3 = 1, v^g = 3 \}$.

Clearly, $av_{I}^1$ and $av_{I}^2$ are only applicable in the intended order. As, after applying $av_{I}^2$, $v^g = 1$ disappears from $v^g$’s memory and can never be reacquired, $av_{I}^1$ and $av_{I}^2$ are deactivated forever after applying them once. All other actions are only applicable after the initialization, i.e., after $av_{I}^2$, due to the new precondition $v^g = 3$. This concludes the proof. ■