A Study of Forward versus Backwards Endgame Solvers with Results in Chinese Checkers

Nathan R. Sturtevant[1] and Abdallah Saffidine[2]

[1] Department of Computer Science
University of Denver
Denver, CO, USA
sturtevant@cs.du.edu
[2] School of Computer Science and Engineering
University of New South Wales
Sydney, Australia
abdallahs@cse.unsw.edu.au

Abstract. When writing an endgame solver that uses retrograde analysis, there are many significant choices that can be made about how to implement the solver. While significant work has been done on building solvers for many games, including Chess and Checkers, we were surprised to find that there has not been a comprehensive study identifying the choice of solver enhancements. This paper represents preliminary work in this direction, exploring several types of forward and backwards solvers, and reporting preliminary results on small versions of Chinese Checkers.

1 Introduction

Endgame databases have been a key component of game-playing programs in games like Checkers [6, 1] and Chess [8, 5], where they contain precise information that is easier to pre-compute than it is to summarize in heuristics or machine-tuned evaluation functions. Despite extensive literature on specific endgames, there is no definitive work describing the different ways that endgame solvers can be built with the trade-offs associated with each of these solvers.

Thus, the goal of this paper is to describe a variety of techniques that can be used to build end-game databases. We look at techniques that work backwards from proven positions, techniques that search forward from unproven positions (looking for proven positions), and hybrid approaches. We discuss the trade-offs of each approach theoretically and then validate these results on a small game of Chinese Checkers.

We assume that we are attempting to prove all states in the state space instead of finding small proof trees [4, 2]. For simplicity, we also do not consider the impact of external memory during search. Endgame databases are typically larger than available RAM and must be built and accessed on disk. In this way, there is significant overlap between external-memory techniques used for breadth-first search [3, 9, 7] and the types of techniques necessary to build endgame databases in memory. Given this complexity, this is left for future work.

2 Background

There are three primary approaches that can be used to build an endgame solver. We roughly classify these into backwards solvers, forward solvers, and hybrid approaches.

2.1 Informal Description

The primary operation in retrograde search is to expand states and propagate proven values (wins and losses) throughout the state space. We classify approaches according to the choice of states that are expanded. A *backwards* solver is one that expands and finds the predecessors of proven states, checking to see if the parents of these states can be proven [8]. A *forward* solver is one that expands unproven states, checking to see if the state can be proven given its current children. We can also consider a *hybrid* variant of the forward solver, combining ideas of the two approaches. A hybrid solver can, for instance, use the general forward approach in most cases, but immediately propagate wins (at min nodes) or losses (at max nodes) backwards [6].

2.2 Model

Definition 1. *A* two-player normal-play game *is a tuple* $\langle \Sigma, \rightarrow, \tau \rangle$ *where* Σ *is a set of states,* $\rightarrow \subseteq \Sigma \times \Sigma$ *is a transition relation, and* $\tau : \Sigma \rightarrow \{\max, \min\}$ *is a turn function.*

For any pair of states s, s', we write $s \rightarrow s'$ to indicate that there is transition from s to s'. We partition the set of states according to whose turn it is $\Sigma_{\max} = \{s \in \Sigma, \tau(s) = \max\}$ and $\Sigma_{\min} = \{s \in \Sigma, \tau(s) = \min\}$.

The expression *normal-play* refers to the understanding that a player loses when it is their turn but they do not have any possible transition. This assumption is very common in Combinatorial Game Theory and often allows for more elegant formal treatments. Keeping in mind that the transition function of other game representations can easily be adapted to fit this formalism, we will also adopt this convention.

The game-theoretic outcome of a state can then be defined as the least fixpoint of the following relations. A max state is *won* if it has a won successor and a min state is won if all its successors are won. Similarly, a max state is *lost* if all its successors are lost and a min state is lost if it has a lost successor. We call *drawn states*, states that are neither lost nor won. We use the mapping $\nu : \Sigma \rightarrow \{\text{win}, \text{loss}, \text{draw}\}$ to denote the outcome of states. The *height* or distance to mate is $\delta : \Sigma \rightarrow \mathbb{N} \cup \{\infty\}$ with $\mathbb{N}$ being the set of non-negative integers. We have $\delta(s) = \infty$ exactly when $\nu(s) = \text{draw}$. These notions can be formalized as follows.

Definition 2. *The set of* winning states $W \subseteq \Sigma$*, the set of* losing states $L \subseteq \Sigma$*, and the* height δ *are defined through the following equations.*

$$\mathcal{W}_{-1} = \mathcal{L}_{-1} = \emptyset \tag{1}$$

$$\begin{aligned}\mathcal{W}_{i+1} =& \{s \in \Sigma_{\max}, \exists s \to s', s' \in \mathcal{W}_i\} \cup \\ & \{s \in \Sigma_{\min}, \forall s \to s', s' \in W_i \wedge \exists s \to s', s' \in \mathcal{W}_i\}\end{aligned} \tag{2}$$

$$\begin{aligned}\mathcal{L}_{i+1} =& \{s \in \Sigma_{\max}, \forall s \to s', s' \in L_i \wedge \exists s \to s', s' \in \mathcal{L}_i\} \cup \\ & \{s \in \Sigma_{\min}, \exists s \to s', s' \in \mathcal{L}_i\}\end{aligned} \tag{3}$$

$$W_i = \bigcup_0^i \mathcal{W}_i \tag{4}$$

$$L_i = \bigcup_0^i \mathcal{L}_i \tag{5}$$

$$W = \bigcup_i \mathcal{W}_i \tag{6}$$

$$L = \bigcup_i \mathcal{L}_i \tag{7}$$

$$\delta(s) = \begin{cases} \min\{i, s \in \mathcal{W}_i \cup \mathcal{L}_i\} & \textit{if } s \in W \cup L \\ \infty & \textit{otherwise} \end{cases} \tag{8}$$

In this formalism, drawn states are only possible if the state space is infinite or admits cycles. Here the set $\mathcal{W}_i$ (resp. $\mathcal{L}_i$) contains the states that can force a win (resp. loss) in exactly i moves, while the set W_i (resp. L_i) contains the states in which max (resp. min) can force a win (resp. loss) in i moves or fewer. Note that this definition restricts a state to only be in $\mathcal{W}_i$ for the single lowest value of i. We write this restriction explicitly in Equations 2 and 3, later we will assume this without writing it explicitly.

In practice, ν and δ are initially unknown and the goal of this paper is to propose and compare algorithms to compute them. Our comparison will be in terms of theoretical complexity and experimentally with the Chinese Checkers domain.

2.3 Metrics

As usual in search problems, the relative performance of solving algorithms varies from one domain to the next. We will therefore characterize the worst-case complexity of the algorithms we consider in terms of state space features such as the number of states or the branching factor. Besides the state space features, the relative cost of algorithmic atomic operations such as applying a move or generating the list of legal transitions may also depend on the domain. For example, applying a move in a specialized implementation of Hex is typically easier than checking for termination whereas computing the next state in General Game Playing is harder than determining if a state is terminal.

In addition to the *number of states* $|\Sigma|$, the maximum height $\delta_{\max}$, we will also use the *number of winning states* $|W|$, the *number of losing states* $|L|$, as well as the following branching factor quantities.

Definition 3. *For each state s, we call* forward branching factor *the number of outgoing transitions: $f(s) = |\{s', s \to s'\}|$ and the* backward branching factor *the number of incoming transitions: $b(s) = |\{s', s' \to s\}|$. The average forward branching factor of a game is then $f_{\text{avg}} = \frac{\sum_{s \in \Sigma} f(s)}{|\Sigma|}$ and the maximal branching factor is $f_{\max} = \max_{s \in \Sigma} f(s)$. The* average *and* maximal backward branching factors b_{avg} *and* $b_{\max}$ *are defined by replacing $f(s)$ with $b(s)$.*

The maximum forward and backward branching factors can often be computed exactly or at least they can be upper-bounded with a domain-specific analysis. In Hex on size $n \times n$, for instance, the maximum forward branching factor is reached in the initial state where each cell of the board is empty, so $f_{\max} = n^2$. The maximum backward branching factor is reached when the board is full and the last move could have been placing any piece of the non-turn player, so $b_{\max} = \frac{n^2}{2}$. The average branching factor, on the other hand, usually needs to be estimated empirically.

While our formal model does not need to represent actions, a fine-grained analysis of algorithm performances can take advantage of distinguishing between the computation of the legal transitions, the *state expansion*, and computing the state corresponding to a given transition, the *action application*. Formally, we will consider the following atomic operations in our analysis.

Definition 4. *On the one hand, a* forward expansion *(resp.* backward expansion*) consists in generating a list of actions from (resp. to) a given state. That is a list of implicit representations of each possible transition. On the other hand, applying or* doing *(resp.* undoing*) an action consists in computing explicitly the state resulting (resp. originating) from the specific transition. We denote the number of atomic forward expansions, backward expansions, action doings, and action undoings, by respectively a_{forw}, a_{back}, a_{do}, and a_{undo}.*

2.4 Naive Retrograde Analysis

Indeed, an easy observation is that for all i, we have $W_i \subseteq W_{i+1}$ and $L_i \subseteq L_{i+1}$. This means that as soon as the state space Σ is assumed to be finite, there exists a finite rank $\delta_{\max}$ such that $W = W_{\delta_{\max}}$ and $L = L_{\delta_{\max}}$. The notation for this rank is justified as we can take $\delta_{\max} = \max\{\delta(s), s \in W \cup L\}$.

These observations give rise to a simple backward induction algorithm to compute the outcome of each state. Algorithm 1 is a direct implementation of lines 2 and 3 of Definition 2 and can be run to determine which states are winning, its dual tells us which states are losing, and the remaining states are drawn.

Algorithm 1: Pseudo code for the Naive solving algorithm

```
1  naive-check(state s, int i)
2      foreach forward action m do
3          s' ← do(s, m)
4          if τ(s) = max ∧ s' ∈ W_i then return ⊤
5          if τ(s) = min ∧ s' ∉ W_i then return ⊥
6      return τ(s) ≠ max

7  naive()
8      i ← −1
9      W_{−1} ← ∅
10     repeat
11         i ← i + 1
12         W_i ← ∅
13         foreach s ∈ Σ do
14             if naive-check(s, i − 1) then W_i ← {s} ∪ W_i
15     until W_i = W_{i−1}
```

Proposition 1. *If the state space is finite and* $\delta_{\max}$ *is the highest height, then the complexity of Algorithm 1 in terms of state expansions and action applications is in the best case* $(\delta_{\max}+1)|\Sigma|(a_{\text{forw}}+a_{\text{do}})$ *and in the worst case* $(\delta_{\max}+1)|\Sigma|(a_{\text{forw}}+f_{\text{avg}}a_{\text{do}})$.

2.5 Model Refinements

In practice, we may not require the exact distance to a win; we may just want to prove the wins (and potentially the losses) as quickly as possible. In this formulation the order in which states are considered matters. We enumerate all states in order $\prec$. For each state, we can compute the current turn player, the list of legal actions, and the list of actions that could have led to this state.

Definition 5. *For a given total ordering on states $\prec$, the* $\prec$-height *is defined through the following equations.*

$$\mathcal{W}_{-1}^{\prec} = \mathcal{L}_{-1}^{\prec} = \emptyset \tag{9}$$

$$\mathcal{W}_{i+1}^{\prec} = \begin{Bmatrix} s \in \Sigma_{\max}, & \exists s \to s', s' \in \mathcal{W}_i^{\prec} \vee (s' \in \mathcal{W}_{i+1}^{\prec} \wedge s' \prec s) \\ s \in \Sigma_{\min}, & \forall s \to s', s' \in \mathcal{W}_i^{\prec} \vee (s' \in \mathcal{W}_{i+1}^{\prec} \wedge s' \prec s) \end{Bmatrix} \tag{10}$$

$$\mathcal{L}_{i+1}^{\prec} = \begin{Bmatrix} s \in \Sigma_{\max}, & \forall s \to s', s' \in \mathcal{L}_i^{\prec} \vee (s' \in \mathcal{L}_{i+1}^{\prec} \wedge s' \prec s) \\ s \in \Sigma_{\min}, & \exists s \to s', s' \in \mathcal{L}_i^{\prec} \vee (s' \in \mathcal{L}_{i+1}^{\prec} \wedge s' \prec s) \end{Bmatrix} \tag{11}$$

$$W_i^{\prec} = \bigcup_0^i \mathcal{W}_i^{\prec} \tag{12}$$

$$L_i^{\prec} = \bigcup_0^i \mathcal{L}_i^{\prec} \tag{13}$$

$$\delta^{\prec}(s) = \begin{cases} \min\{i, s \in \mathcal{W}_i^{\prec} \cup \mathcal{L}_i^{\prec}\} & \text{if } s \in W \cup L \\ \infty & \text{otherwise} \end{cases} \tag{14}$$

Note that Equation (10) and Equation (11) are recursive but not circular because $\prec$ is total ordering. It is indeed possible to determine whether a state s belongs to $\mathcal{W}_{i+1}^{\prec}$ solely based on $\mathcal{W}_i^{\prec}$ and on which states smaller than s according to $\prec$ belong to $\mathcal{W}_{i+1}^{\prec}$. Additionally note that we assume a state only appears in the first possible $\mathcal{W}_i^{\prec}$ or $\mathcal{L}_i^{\prec}$ respectively, but for clarity in seeing the nature of the total ordering we omit the additional logic required to specify this precisely.

The next proposition shows how the $\prec$-height and the sets $\mathcal{W}_i^{\prec}$ $\mathcal{L}_i^{\prec}$ can be formally compared to the ordering-independent versions. The result is proved by induction on i.

Proposition 2. *For any ordering $\prec$, for any height i, we have $W_i \subseteq W_i^{\prec}$ and $L_i \subseteq L_i^{\prec}$, and for any state $s \in \Sigma$, we have $\delta^{\prec}(s) \leq \delta(s)$.*

An ordering $\prec$ is *perfect* if any state has $\prec$-height 0. This means that a single-pass run of the forward search algorithm is sufficient to make all possible deductions.

Proposition 3. *If an ordering $\prec$ is consistent with the height, then it is perfect. That is, if for each transition $s \to s'$ we have $\nu(s) = \nu(s') \wedge \delta(s') < \delta(s) \implies s' \prec s$, then $\forall s, \delta^{\prec}(s) = 0$.*

To the standard metrics given in Definition 3, we add

Definition 6. *For a given heuristic state ordering $\prec$, we will also use the* maximum $\prec$-height*, $\delta_{\max}^{\prec} = \max\{\delta^{\prec}(s), s \in W \cup L\}$.*

As we will see later, the maximum $\prec$-height quantifies the quality of the state ordering and orderings with smaller maximum height are to be preferred.

3 Algorithms and Variants

In the following pseudo-code we assume that each state has a number of other properties associated with it that provide meta-information about the solving. For instance, s.solved, indicates that a state has been completely solved, s.recent means that it has been set in the current iteration, s.depth is the distance to win or loss, and s.visited is used to indicate whether the backwards check procedure has already been performed on a state. These and other properties are described in the following sections.

3.1 One-step Forward Check

We first describe the `forward-check` procedure given in Algorithm 2. It acts as a subroutine for both the forward and backward search algorithms. `forward-check` take a state s as argument and determines whether it can be solved based on the current information we have on the children of s. The `recent` property indicates whether the state was changed in the current iteration. The subroutine also uses two meta-parameters, `win-only` and `layered`. When the parameter `win-only` is set, `forward-check` only attempts to prove that s is a max win. Setting the `layered` parameter indicates that we desire accurate distance-to-mate information.

Algorithm 2: Pseudo code for the Forward check procedure

```
1  forward-check(state s)
2      d ← 0
3      foreach s → s' do
4          // Cannot prove win at min node if one child unknown
5          if win-only ∧¬s'.solved ∧ ¬s.maxturn then return
6          // One child unsolved or changed in current iteration
7          if ¬s'.solved ∨ (layered ∧ s'.recent) then d ← ∞
8          else if s.maxturn ∧ ¬s'.win ∧ ¬win-only then d←max(d, 1+s'.depth)
9          else if ¬s.maxturn ∧ s'.win then d←max(d, 1+s'.depth)
10         else
11             // proven outcome for current player
12             s.solved ← ⊤
13             s.win ← s.maxturn
14             s.depth ← 1 + s'.depth
15             return
16     if d < ∞ ∧ (¬win-only ∨ ¬s.maxturn) then
17         s.solved ← ⊤
18         s.win ← ¬s.maxturn
19         s.depth ← d
```

To do so, we use an auxiliary variable d to represent to represent the distance-to-mate in case s is not a win for $\tau(s)$. The value $d = \infty$ indicates that at least

one child of s is not known to be losing for $\tau(s)$. The algorithm traverses every child s' of s (Line 3) and tests if the information contained in s' is helpful to settle s. In the `win-only` case, if it is min's turn in s and s' is not proven to be a max win, then we will not be able to prove a max win in s (Line 5). Else, if s' is not solved yet, or if the distance-to-mate information is unreliable, then we will not be to prove a $\tau(s)$ loss in this call to `forward-check` (Line 7). Otherwise, if s' is a proven loss for $\tau(s)$, we can update the auxiliary variable d (Lines 8 and 9). In the remaining case, s' is a proven win for $\tau(s)$ with accurate-enough distance-to-mate information and the call to `forward-check` is completed. If the subroutine has not been exited before each child has of s been visited, then we might have been able to prove that s was a $\tau(s)$ loss (Line 16).

For each solved state s, we use the s`.win` bit field to record if s is a max win. If the `win-only` parameter is set, that is, we do not intended on discriminating between draws and losses, then the `.win` field can be discarded and Algorithm 2 can be simplified to omit Lines 13 and 18. Similarly, accurate distance-to-mate information is only recorded if `layered` is set. If it is not, then the `.depth` field can be discarded and we can omit Lines 14 and 19.

3.2 Direct Backward Propagation

Looking at Equations (2) and (3), we see that there are two different conduits for the outcome of a specific state to be settled. The universal condition requires all children state to be proven loss to allow an inference on the current state, the existential condition, on the other hand, is met as soon as any child state is a proven win. The dual to this observation is that solving a state might in some cases allow an immediate solving of the parent states. This idea leads to a variant of retrograde analysis where as soon as a state s is solved, the parents of the state are checked to see if an immediate solution can be deduced from s.

The `quick-check(`s`)` procedure in Algorithm 3 takes a solved state s as input and implements this idea. Any unsolved parent s' of s such that player $\tau(s')$ wins in s can be updated as solved without checking its other children. In practice, if the domain has strict turn alternation, then the turn check in Line 3 can be factored out of the loop.

Algorithm 3: Pseudo code for the Quick check algorithm

```
1 quick-check(state s)
2     foreach s' → s do
3         if s'.maxturn = (win-only ∨ s.win) then
4             if ¬s'.solved then
5                 s'.solved ← ⊤
6                 s'.win ← s.win
7                 s'.depth ← 1 + s.depth
8                 s'.recent ← ⊤
```

Again, depending on the value of the meta-parameters `win-only` and `layered`, the fields `.win`, `.depth` and `.recent` can be discarded, and Lines 6 to 8 can be omitted.

3.3 Forward Search

We can now define a forward search method building on Algorithm 2 and 3. It consists of the pseudo-code given in Algorithm 4. The main loop (Line 3) is run until a fixpoint is reached, which we detect by tracking changes in iterations in the auxiliary variable c. For each pass of the main loop, we traverse all unsolved states s and attempt to settle them by using the `forward-check` subroutine. If the solving status of s changes, then we can also attempt to directly solve parents of s using the `quick-check` subroutine.

Algorithm 4: Pseudo code for the Forward search algorithm

```
1  forward()
2      c ← ⊤
3      while c do
4          c ← ⊥
5          foreach s ∈ Σ do
6              s.recent ← ⊥
7              if not s.solved then
8                  forward-check(s)
9                  if s.solved then
10                     c ← ⊤
11                     s.recent ← ⊤
12                     if direct then quick-check(s)
```

The call to the latter subroutine is only enabled when the `direct` parameter is set. Just as in the previous algorithms, if `layered` is not set, we can discard the s`.recent` field and omit Lines 6 and 11.

3.4 Backward Search

The backward search algorithm can also be decomposed into a subroutine, `backward-check` (Algorithm 5), which is built on `forward-check`, and main loop, Algorithm 6.

The `backward-check` procedure takes a solved state s as input and attempts to solve the parents of s. To do so, for unsolved parent s', we first check whether the solution to s can be directly use to settle s' in a similar fashion to the `quick-check` procedure (Line 4). If not, then a full verification via the siblings of s is required, and a called to `forward-check` performs it (Line 9. If any parent s' is thus solved, the information is recorded and passed back to the

Algorithm 5: Pseudo code for the Backward check procedure

```
1  backward-check(state s, bool c)
2      foreach s' → s do
3          if ¬s'.solved then
4              if s'.maxturn = (s.win ∨ win-only) then
5                  s'.solved ← ⊤
6                  s'.win ← s.win
7                  s'.depth ← 1 + s.depth
8              else
9                  forward-check(s')
10             if s'.solved then
11                 c ← ⊤
12                 if direct then quick-check(s')
```

caller via variable c (Line 11), and we have the possibility to attempt a direct backpropagation to the parents of s' (Line 12).

If the distinction between max losses and draws is not made or the distance-to-mate information is not preserved, then Lines 5 and 6 can be omitted respectively.

The backward search method is explicitly in Algorithm 6. Similar to the forward search method, it involves a fixpoint computation (Line 8) and a called to the corresponding subroutine `backward-check` on each relevant state (Line 12). However, the states on which we call `backward-check` are the solved ones, whereas the forward search method needed to call `forward-check` one states yet to be solved. A possible optimization of this approach is to only call `backward-check` a single time per state, and to do so when the state is solved for the first time. In that case, the parameter `no-dup` is set and the bit field s`.visited` captures whether `backward-check` has already been called on s (Line 13). Another difference between the two search methods is that `backward` first needs to identify and solve the terminal states (Lines 2 to 6).

The main body of the algorithm is two nested loop that roughly correspond to those of `naive`. The first difference between `naive` and `forward` is that the latter only calls the embedded subroutine if the current state is not solved (Line 7). The second difference is that while `naive-check` was explicitly using the previous layer of solved states, `forward-check` can take advantage of states solved in much earlier iterations without recomputing them and can also use states already solved in the current iteration.

4 Theoretical Analysis

The memory needed for these algorithms depends on which parameterization of the algorithms we need and kind of state-information we would like to preserve. Two bits per state are needed when computing the outcome class whereas a single bit is sufficient when we only compute the winning states, so `win-only` saves a

Algorithm 6: Pseudo code for the Backward search algorithm

```
1  backward()
2      foreach state s ∈ Σ do
3          if s is terminal then
4              s.solved ← ⊤
5              s.win ← ¬s.maxturn
6              s.depth ← 0
7      c ← ⊤
8      while c do
9          c ← ⊥
10         foreach s ∈ Σ do
11             if s.solved ∧ (¬no-dup ∨ ¬s.visited) then
12                 backward-check(s, c)
13                 s.visited ← ⊤
```

bit per state. If the height of the game can be bounded by $\delta_{\max} < 2^d$, then storing the distance-to-mate information requires at most d bits per state. Finally, the forward algorithm needs a bit per state for the `layered` parameterization, so as to distinguish states solved in the current iteration from states solved in previous iterations, and the backward algorithm needs a bit per state for the `no-dup` parameterization.

The time complexity of the algorithm also depends on the parameterization. In general, `forward-check` takes at most f_{avg} steps. As for the number of calls to `forward-check`, a rough approximation is that for each iteration of the main loop (Line 3 to Line 11), we will at most call the subroutine $|\Sigma|$ times, giving an overall complexity estimate of $|\Sigma|\delta_{\max} f_{\text{avg}}$.

However, since we are not attempting to re-solve states previously solved (Line 7), more refined estimates of the number of subroutine calls are possible. In turn, this provides us with a better estimate of the number of forward and backward expansions as shown in Table 1. If we make the simplifying assumption that the forward branching factor is uniform across the different layers, then Table 1 provides us with recommendation as to which parameterization to use. The formulas show that if enough memory is available to add a bit per state, then using unsetting `win-only` is always preferable. Together with Prop. 2, we can see that unsetting `layered` also improves the complexity. In conclusion, if computing distance-to-mate is not required and enough memory is available to store two bits per states, then the best approach is to compute win and loss outcomes in a non-layered manner.

Intuitively, enabling `direct` allows earlier proofs of some states at the expense of additional backward expansion. Unfortunately, our formalization is not refined enough to provide an improved bound on the number of forward expansions in that case, and so Table 1 does not reflect our intuition.

A similar analysis reveals loose upper bounds on the number forward and backward expansion in the Backward search algorithm, Tables 2 and 3. Indeed,

Table 1. Upper bounds on the number of forward and backward expansions in Algorithm 4 depending on the parameterization.

`layered`	`direct`	`win-only`	Expansions			
			Forward		Backward	
			Iteration i	Overall (NS: loose)	Iteration i	Overall
Yes	No	Yes	$\lvert\Sigma\rvert - \lvert\mathcal{W}_i\rvert$	$\delta_{\max}\lvert\Sigma\rvert - \lvert W\rvert$		
Yes	No	No	$\lvert\Sigma\rvert - \lvert\mathcal{W}_i \cup L_i\rvert$	$\delta_{\max}\lvert\Sigma\rvert - \lvert W \cup L\rvert$		
No	No	Yes	$\lvert\Sigma\rvert - \lvert\mathcal{W}_i^{\prec}\rvert$	$\delta_{\max}^{\prec}\lvert\Sigma\rvert - \lvert W\rvert$		
No	No	No	$\lvert\Sigma\rvert - \lvert\mathcal{W}_i^{\prec} \cup L_i^{\prec}\rvert$	$\delta_{\max}^{\prec}\lvert\Sigma\rvert - \lvert W \cup L\rvert$		
No	Yes	Yes	$\lvert\Sigma\rvert - \lvert\mathcal{W}_i^{\prec}\rvert$	$\delta_{\max}^{\prec}\lvert\Sigma\rvert - \lvert W\rvert$	$\lvert\mathcal{W}_{i+1}^{\prec}\rvert - \lvert\mathcal{W}_i^{\prec}\rvert$	$\lvert W\rvert$
No	Yes	No	$\lvert\Sigma\rvert - \lvert\mathcal{W}_i^{\prec} \cup L_i^{\prec}\rvert$	$\delta_{\max}^{\prec}\lvert\Sigma\rvert - \lvert W \cup L\rvert$	$\lvert\mathcal{W}_{i+1}^{\prec} \cup L_{i+1}^{\prec}\rvert - \lvert\mathcal{W}_i^{\prec} \cup L_i^{\prec}\rvert$	$\lvert W \cup L\rvert$

each call to `backward-check` results in a backward expansion of the argument state. Additionally, each such call can result in multiple calls to `forward-check` in the parents of the argument state. The number of calls to `forward-check` can be upper-bounded bounded by the number of parents, which averages to b_{avg}.

Table 2. Upper bounds on the number of forward expansions in Algorithm 6 depending on the parameterization.

`no-dup`	`direct`	`win-only`	Forward Expansions	
			Iteration i	Overall
Yes	No	Yes	$b_{\text{avg}}\lvert\mathcal{W}_i\rvert$	$\lvert\Sigma\rvert + b_{\text{avg}}\lvert W\rvert$
Yes	No	No	$b_{\text{avg}}\lvert\mathcal{W}_i \cup \mathcal{L}_i\rvert$	$\lvert\Sigma\rvert + b_{\text{avg}}\lvert W \cup L\rvert$
Yes	Yes	Yes	$b_{\text{avg}}\lvert\mathcal{W}_i^{\prec}\rvert$	$\lvert\Sigma\rvert + b_{\text{avg}}\lvert W\rvert$
Yes	Yes	No	$b_{\text{avg}}\lvert\mathcal{W}_i^{\prec} \cup \mathcal{L}_i^{\prec}\rvert$	$\lvert\Sigma\rvert + b_{\text{avg}}\lvert W \cup L\rvert$
No	No	Yes	$b_{\text{avg}} \sum_{j=0}^{i} \lvert W_j\rvert$	$\lvert\Sigma\rvert + b_{\text{avg}}\delta\lvert W\rvert$
No	No	No	$b_{\text{avg}} \sum_{j=0}^{i} \lvert W_j \cup L_j\rvert$	$\lvert\Sigma\rvert + b_{\text{avg}}\delta\lvert W \cup L\rvert$
No	Yes	Yes	$b_{\text{avg}} \sum_{j=0}^{i} \lvert\mathcal{W}_j^{\prec}\rvert$	$\lvert\Sigma\rvert + b_{\text{avg}}\delta\lvert W\rvert$
No	Yes	No	$b_{\text{avg}} \sum_{j=0}^{i} \lvert\mathcal{W}_j^{\prec} \cup \mathcal{L}_j^{\prec}\rvert$	$\lvert\Sigma\rvert + b_{\text{avg}}\delta\lvert W \cup L\rvert$

5 Experimental Results

We validate the theoretical analysis of the forward and backward solvers with an empirical study on a small-enough-sized variant of Chinese Checkers. This variant of Chinese Checkers involves two players with either 2 or 3 pieces each and competing on a 7×7 board. We report results on both a single-threaded and multi-threaded implementations of the algorithms described in this paper. The experiments are run on a 4-core machine with hyperthreading.

Table 3. Upper bounds on the number of backward expansions in Algorithm 6 depending on the parameterization.

no-dup	direct	win-only	Backward Expansions	
			Iteration i	Overall (loose)
Yes	No	Yes	$\lvert\mathcal{W}_i\rvert$	$\lvert W\rvert$
Yes	No	No	$\lvert\mathcal{W}_i \cup \mathcal{L}_i\rvert$	$\lvert W \cup L\rvert$
Yes	Yes	Yes	$\lvert\mathcal{W}_i\rvert + b_{\text{avg}}\lvert\mathcal{W}_{i+1}\rvert$	$(1 + b_{\text{avg}})\lvert W\rvert$
Yes	Yes	No	$\lvert\mathcal{W}_i \cup \mathcal{L}_i\rvert + b_{\text{avg}}\lvert\mathcal{W}_{i+1} \cup \mathcal{L}_{i+1}\rvert$	$(1 + b_{\text{avg}})\lvert W \cup L\rvert$
No	No	Yes	$\lvert W_i\rvert$	$\delta\lvert W\rvert$
No	No	No	$\lvert W_i \cup L_i\rvert$	$\delta\lvert W \cup L\rvert$
No	Yes	Yes	$\lvert W_i\rvert + b_{\text{avg}}\lvert W_{i+1}\rvert$	$(1 + b_{\text{avg}})\delta\lvert W\rvert$
No	Yes	No	$\lvert W_i \cup L_i\rvert + b_{\text{avg}}\lvert W_{i+1} \cup L_{i+1}\rvert$	$(1 + b_{\text{avg}})\delta\lvert W \cup L\rvert$

We also have a space-optimized implementation using no more than 2 bits per state. Although we do not report detailed results for the sake of space and clarity, we have observed that the space-optimized version runs about three times faster than the default implementation, and we conjecture that the time savings are mostly due to better memory locality and fewer runtime options.

5.1 Impact of the Algorithm Parameterization

We start by examining the performance of both algorithmic approaches on 7×7 Chinese Checkers with 2 pieces each. We report the number of forward and backward expansions, the number of iterations of the fixpoint loops appearing in both algorithms, as well as the global time spent.

The performance of Algorithm 4 is given in Table 4. The number of forward expansions is positively correlated by the total time needed by the algorithm. The performance decreases when we only compute wins as opposed to wins and losses, and the performance improves when we drop the layer constraint. This match perfectly the theoretical understanding in Section 4.

Table 4. Forward Search performance on Chinese Checkers size 7×7 with 2 pieces each and 1 thread.

Parameters			Expansions		Iterations	Time (s)
layered	direct	win-only	Forward	Backward		
Yes	No	Yes	67,248,898		37	562.3
Yes	No	No	38,293,120		37	344.5
No	No	Yes	55,444,443		29	467.0
No	No	No	27,112,140		29	250.9
No	Yes	Yes	32,592,724	549,791	17	281.5
No	Yes	No	14,017,955	1,099,582	17	145.0

The performance of Algorithm 6 is given in Table 5. Again, we see that the number of expansions is positively correlated by the total time needed by the algorithm. Note that the `direct` setting does not incur an increase in the number of backward expansion, unlike predicted by the general model of Table 3, because the turn order in Chinese Checkers is strictly alternating and it allows for further domain-specific implementation optimizations. Unlike the forward approach, the time performance of Algorithm 6 improves when we only attempt to identify positions winning for max. This matches indeed the model in Table 2 and 3. On the other hand, the `direct` setting does lead to fewer forward expansions and faster overall time. This was not shown in the worst-case analysis of the previous section but it matches the intuition and points to possible refinements of the formal framework so as to better capture this phenomenon.

Table 5. Backward Search performance on Chinese Checkers size 7×7 with 2 pieces each and 1 thread.

Parameters			Expansions		Iterations	Time (s)
`no-dup`	`direct`	`win-only`	Forward	Backward		
Yes	No	Yes	4,090,505	1,271,256	37	50.8
Yes	No	No	5,780,926	2,542,512	37	67.0
Yes	Yes	Yes	3,374,445	1,271,256	37	48.6
Yes	Yes	No	4,348,806	2,542,512	37	61.9

5.2 Scalability

We did not describe in pseudo-code how to parallelize Algorithm 4 and 6 for the sake of simplicity. Still, we implemented multi-threaded versions in a relatively direct fashion and we can observe the performance of both algorithms and their parameterization as a function of the number of threads available for computation. Specifically, Table 6 reports solving time for the sequential and the 8-thread versions.

From these preliminary results, we extract that although all algorithm parameterizations seem to benefit from increased computational power in the form of threads, it seems that the forward approach scales better than the backward approach. We conjecture that this behavior can be traced back to the fact that it would be easier to split Line 5 in Algorithm 4 across multiple threads in a fair way than splitting Line 10 in Algorithm 6.

The results in Table 6 indicate that both approaches perform at a comparable level with 8 threads as long as the right parameters are chosen. We now attempt to solve a larger domain with similar properties: 7×7 Chinese Checkers with 3 pieces each. For Algorithm 4, we set `layered` to false, `direct` to true, and `win-only` to false, consistently with the seemingly best approach according to intuition and to Table 6. The total time needed for solving this larger domain

Table 6. Solving time for Forward and Backward Search with varying number of threads on Chinese Checkers size 7×7 with 2 pieces each.

Search	Parameters			Solving time (s)	
	`layered`	`direct`	`win-only`	1 thread	8 threads
Forward	Yes	No	Yes	562.3	125.4
	Yes	No	No	344.5	72.9
	No	No	Yes	467.0	93.7
	No	No	No	250.9	45.7
	No	Yes	Yes	281.5	64.3
	No	Yes	No	145.0	31.4
	`no-dup`	`direct`	`win-only`	1 thread	8 threads
Backward	Yes	No	Yes	50.8	33.1
	Yes	No	No	67.0	48.5
	Yes	Yes	Yes	48.6	32.6
	Yes	Yes	No	61.9	47.7

is 14,555s. Similarly, for Algorithm 6, we set `no-dup` to true, `direct` to true, and `win-only` to true. The total solving time for the backward approach is then 14,322s.

According to these results, both approaches seem to scale in a consistent manner with the domain size. The solving times are remarkably close for Chinese Checkers, but one may still prefer an approach over the other depending on the experimental resources and needs: the backward approach uses only half as much memory whereas the forward approach distinguishes between draws and losing positions.

6 Conclusion

We have investigated different approaches to endgame solvers based on retrograde analysis and described a couple natural optimizations. A formal model of the state space allowed us to quantify the impact of these optimizations on the worst-case complexity of the solving algorithms. We managed through our theoretical analysis to formally justify some of our intuitions as to which set of optimizations was most beneficial, but the model was not rich enough to provide a complete picture. To complement the theoretical examination, we performed an empirical study on the Chinese Checkers domain. We used a small-size variant to compare all parameter settings and obtained results that match the intuition and the theoretical analysis. We also investigated the extent to which the algorithms could scale with additional resources in the form of a multi-threaded implementation and scale to a medium-size variant of Chinese Checkers.

In conclusion, our results shed light on the forward and the backward approach for endgame solving and demonstrate that both have merits, even in the specific case of Chinese Checkers.

References

1. Björnsson, Y., Schaeffer, J., Sturtevant, N.R.: Partial information endgame databases. In: van den Herik, H.J., Hsu, S., Hsu, T., Donkers, H.H.L.M. (eds.) 11th International Conference on Advances in Computer Games, (ACG) 2005. Revised Papers. Lecture Notes in Computer Science, vol. 4250, pp. 11–22. Springer, Taipei, Taiwan (2006)
2. Buro, M., Long, J.R., Furtak, T., Sturtevant, N.R.: Improving state evaluation, inference, and search in trick-based card games. In: IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009. pp. 1407–1413 (2009)
3. Korf, R.E.: Best-first frontier search with delayed duplicate detection. In: McGuinness, D.L., Ferguson, G. (eds.) Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence (AAAI). pp. 650–657. AAAI Press / The MIT Press, San Jose, California, USA (2004)
4. Moldenhauer, C., Sturtevant, N.: Optimal solutions for moving target search. In: Autonomous Agents and Multiagent Systems (AAMAS). pp. 1249–1250. International Foundation for Autonomous Agents and Multiagent Systems (2009)
5. Nalimov, E., Haworth, G.M., Heinz, E.A.: Space-efficient indexing of endgame tables for chess. ICGA Journal 23(3), 148–162 (2000)
6. Schaeffer, J., Björnsson, Y., Burch, N., Lake, R., Lu, P., Sutphen, S.: Building the checkers 10-piece endgame databases. Advances in Computer Games 10, 193–210 (2003)
7. Sturtevant, N., Rutherford, M.: Minimizing writes in parallel external memory search. International Joint Conference on Artificial Intelligence (IJCAI) (2013)
8. Thompson, K.: Retrograde analysis of certain endgames. ICCA Journal 9(3), 131–139 (1986)
9. Zhou, R., Hansen, E.A.: Parallel structured duplicate detection. In: Twenty-Second AAAI Conference on Artificial Intelligence (AAAI). pp. 1217–1224. AAAI Press, Vancouver, British Columbia, Canada (2007)