Faster State Manipulation in General Games using Generated Code

Kevin Waugh
waugh@cs.ualberta.ca
Department of Computing Science
University of Alberta
Edmonton, AB Canada T6G 2E8

Abstract

Many programs for playing games rely on some type of state space search to choose their actions. It is well known that there is a correlation between the depth to which these programs search and the strength of their play. Unfortunately when playing games in the general game-playing competition, manipulating states requires expensive logical inference and as a result, programs that play these games cannot examine many states before they are required to act. Typically, these programs make use of a Prolog package to perform the required logical inference. These packages are not designed specifically for general game playing. As a result, there is great potential for improving the strength of play by designing a specialized package to perform the state manipulation tasks. In this paper we introduce `gdlcc`, a program that takes a general game description and creates a game specific `C++` library for performing state manipulating tasks. Experimental results show that a program using this library, as opposed to a Prolog package, can examine between 60% and 1760% more states per action.

1 Introduction

Most strong computer game playing programs are specifically designed and tuned to play only a single game. For example, *Chinook* [Schaeffer *et al.*, 1992], the world's strongest Checkers playing program, can crush all human players, but it does not have the ability to play Chess despite the similarities between the two games. Both these games are played on an eight by eight *board* that is occupied by *pieces*. Pieces *move* across the board and can *capture* the opponent's pieces. These concepts are all programmed into Chinook, but, unlike a human, the program cannot use its knowledge of these concepts to adapt to a different game. The General Game Playing Competition [Genesereth *et al.*, 2005], which has been run at AAAI since 2005, is an attempt to spur Artificial Intelligence research into solving these types of problems. Competitors write programs that can play any game that can be described in the *Game Description Language* (GDL) [Love *et al.*, 2008]. Since GDL can describe both Chess and Checkers, a general game playing program can not only play both games, but can potentially use its knowledge and past experiences from one game to its advantage when playing the other.

Strong game-playing programs aim to maximize their reward in the presence of one or more adversaries. Often these programs attempt to model the play of their opponents in order to exploit their weaknesses. That is, how the opponent will play is often unknown and a program must reason and act under this uncertainty. This problem is still present in general games, but many additional problems arise under this new framework. One important problem is how to represent the game in memory. Much of the strength of strong computer players comes from the speed and efficiency afforded by custom data structures. In general games we do not have this luxury available to us. Having a program determine a smart way of representing an arbitrary game and relating this representation to previous games and known concepts is a large obstacle.

Though these problems appear quite daunting at first, some progress has been made since the competition's inception. Recent competitors are visibly stronger than their predecessors. As some type of search typically underlies most general game playing programs, much of the recent progress in this area has been due to searching in a more intelligent fashion. Since search is typically guided using a heuristic function, a more informed heuristic can focus the search on more important areas of the state space. For this reason, evaluating the effectiveness of different heuristics can be useful for increasing the strength of a program [Clune, 2007]. Similarly, automatically extracting features and heuristics has also proven to be useful [Utgoff, 2001; Kuhlmann *et al.*, 2006; Schiffel and Thielscher, 2007a; Kaiser, 2007]. Other work on transferring heuristic knowledge from one game to another has also shown promise [Banerjee *et al.*, 2006; Banerjee and Stone, 2007; Taylor *et al.*, 2007; Kuhlmann and Stone, 2007].

Though the strength of entrants to the competition is increasing due to these advancements, the play is still far from world class. This is unlikely to change in the near future as progress will continue to be made in small steps as opposed to giant leaps. This slow progression has an unfortunate side effect as one research goal from year to year is merely to win the competition. Surely, steps towards the ultimate goal of strong general intelligence would help a program's chance of winning, but it is not necessarily the case that strengthening the field of competitors yearly means that we are in-

deed moving towards improved general intelligence. Indeed, in this paper we present an enhancement that can be used by any general game playing program to strengthen its play, but this enhancement does not provide us with any new insight towards solving the larger problems in Artificial Intelligence.

As the successful general game playing programs rely on search, we can take advantage of the well known correlation between the strength of play and the amount of the state space that is examined per turn. That is, without altering the underlying heuristic, increasing the amount of the state space examined each turn will likely improve the quality of play of a program. Unfortunately as a side effect of the flexibility that GDL affords us, state manipulation in GDL is extremely slow as it requires expensive logical inference. Most competitors make use of one of the various generic Prolog packages to perform this inference. These packages are not tailored in any way towards the specific game being played and often come with additional capabilities that are not needed by the players. In this paper we introduce `gdlcc`, a `C++` program that creates a custom `C++` library from a general game description. This library replaces the need for a bulky Prolog package as it has a common interface for performing state manipulation tasks. Before delving into the workings of our system, we will briefly overview the past competitions. We will then expand on the state manipulation tasks that require logical inference. After, we will describe how `gdlcc` programatically generates `C++` code to perform these tasks and some minor optimizations that it uses during this process. Finally, we will show experimental results contrasting `gdlcc` to `YAP` [CRACS and LIACC, 2008], a Prolog package, in terms of their abilities to perform state manipulation tasks.

2 Background

The winner of the first General Game Playing Competition was Cluneplayer [Clune, 2007]. Cluneplayer attempts to create an accurate heuristic function by automatically identifying features for a particular game from a base set of features common in many games. For example, the concept of pieces and mobility are known to Cluneplayer and it attempts to relate these concepts to the logical description of the game to automatically create a set of potential features. Using these potential features, Cluneplayer creates and evaluates many different heuristic functions to find one of suitable quality. This heuristic is then used during play in conjunction with an Alpha-Beta search. The winner of the second competition, Fluxplayer [Schiffel and Thielscher, 2007b], employees a similar tactic of automatically constructing and evaluating a heuristic function. In 2007 and 2008, CADIA-Player [Finnsson, 2007] won the competition using a different approach. Instead of identifying a heuristic function, it employes a Monte Carlo simulation-based evaluation function in conjunction with the UCT algorithm [Finnsson and Björnsson, 2008; Kocsis and Szepesvári, 2006]. Recently, this approach has been quite successful in computer Go. The beauty of this approach is that UCT guides the search in the proper direction given that enough simulations can be done to make the evaluation function accurate.

With the success of CADIA-Player, many other teams have incorperated using simulations in their evaluation functions. More specifically, one simulation refers to playing out the remainder of the game using some set policy (such as, all moves chosen randomly) and the terminal value is taken to be the state's value. In this framework, there are two fundamental ways to improve the quality of a simulation-based player. First, one can simply do more simulations to get a better estimation of the true underlying value of a state. Second, one can use heuristics to improve the policy used during the simulations [Sharma *et al.*, 2008]. Here, the simulation policy is more intelligent or informed. For example, the *history heuristic* will give preference in the simulation to moves that have shown more success in the past [Schaeffer, 1983].

With the simulation-based approaches, it is perhaps even more important that the state manipulation tasks be done quickly, as the evaluation function itself might be required to perform many further state manipulations. Specifically, the state manipulation tasks that require logical inference are: evaluating whether or not the game is over and evaluating the utility of each player in the case that it is, evaluating which actions are legal for each player from a state, and computing a successor state given the joint actions of all players. Each of these tasks are computed using simple logical inference given the rules described in GDL. The advantage to using Prolog for this inference is that it is relatively simple to construct a player in this fashion. The translation from GDL to Prolog is straightforward. Also, many of the powerful Prolog engines have bindings into the most popular languages, which allows for the engine to be easily embedded inside another program. These Prolog packages do many optimizations behind the scenes to make the logical inference as efficient as possible. In fact, some compile the inference rules into an efficient bytecode representation. Despite these extra optimizations, using one of these packages will cause a general game playing program to inherit a lot of unneeded overhead as the machinery provided by Prolog is much more sophisticated than we require. For example, a Prolog package may allow for the logical inference rules to be created, modified or removed. In a general game, these logical inference rules are fixed over the duration of play. Also, a Prolog package can allow for many different *queries*. In a general game, only four top-level queries are ever performed and the subqueries that they depend on are again fixed. Furthermore, Prolog also has additional operators and functions that are not present in GDL, such as the *cut* operator, console print statements, or math operators and functions like $\sin$ or $\cos$.

3 System Description

To overcome the unnecessary overhead of Prolog, we can instead resort to a custom solution for our logical inference needs. Creating a GDL interpreter of sorts would be far slower than using a Prolog package, but generating code that can be compiled into machine code is certainly feasible at first glance. During play, only *facts* change from state to state. Since the logical inference rules are not required to change and the top-level queries do not vary between states, the query resolution rules can be hard-coded. The way these rules can be hard-coded is not unique and the performance of

the required inference can be drastically effected by a good or poor choice in this matter. It is also possible that the best resolution path might depend on the actual game state currently being examined, which would put a hard-coded solution, at a disadvantage to a smarter dynamic one. Even with these potential issues and room for further improvement, we will start with a naive hard-coded solution as currently our only goal is to merely beat a generic Prolog package.

Before engineering our solution, we need a brief understanding of how a game's state is represented. A general game description is made up of *facts* and *inference rules*. Facts are *grounded relations* that are known to be true. Inference rules make use of facts and inference rules to *prove* the truth of other relations. The state of a game is simply all relations that are true given a set of facts. When players make actions, the facts change and hence the set of relations that are true also change. For example, a game might have a relation `day` that is true in the initial state and the inference rule `(<= sunny day)`. This would mean that `sunny` is also true in the initial state. Relations can also have a fixed number of *literal* or *functional* arguments. For example, if we have `(open door1)`, then `open` is the name of a relation with a single argument, which happens to be the literal `door1`. We could also have `(cell 1 2 (piece queen))` and `(cell 3 4 empty)`. Here, `(piece queen)` is a functional argument named `piece` with a single argument `queen`. When we wish to infer something from a set of relations and inference rules, we call the process a *query*. Continuing with our first example, we could query `sunny` and the system would use the inference rules to see that the relation is indeed true in the initial state. We can also query relations with *unknowns* in the place of arguments. Here, we are asking the inference system to return all substitutions that would make the relation true. For example, the query `(door ?x)` would result in `(door open1)` and the query `(cell ?x ?y ?z)` would result in {`(cell 1 2 (piece queen))`, `(cell 3 4 empty)`}. The query `(cell 1 ?x ?y)` or `(cell ?x ?y (piece ?z))` would only return `(cell 1 2 (piece queen))`. We call the process of finding all true substitutions *resolution*. Resolution is a fairly straightforward recursive procedure. First, we can look through any facts we know about the current relation and check for possible substitutions that match our current query. Second, we look through the list of inference rules to find rules that could imply something that matches our current query. We recursively resolve the body to find matches. For example, if we have the query `(unlocked ?x)` and the implication rule `(<= (unlocked ?x) (open ?x))` we find that `(open door1)` matches the body and therefore `(unlocked door1)` is also true. The GDL specification ensures that any valid game can have any query resolved using this procedure.

For GDL specifically, there are some special relations that are important for understanding and manipulating the game. For example, the relation `(role ?x)` defines all the players in the game and it cannot be implied by an inference rule. The inference system that we are creating needs merely a way of keeping track of the mutable facts and a way to query these special relations in order to perform its required tasks. The mutable facts that change between states all have the form `(true ?x)`. The facts that are true in the initial state are denoted `(init ?x)` in the GDL description and one can find which facts are true in a successor state by querying the `(next ?x)` relation. When querying the `(next ?x)` relation, one must set the appropriate `(does ?role ?action)` facts to tell the system how each player has acted from the state. To evaluate which actions are legal from a state, the system queries the `(legal ?role ?action)` relation. The `terminal` relation is true in states where the game has ended and in those states querying the `(goal ?role ?value)` relation determines the reward to each player. We see that only a few queries are required for an inference system to interact with GDL. Specifically, our system must be able to query `(init ?x)`, `terminal`, `(goal ?role ?value)`, `(legal ?role ?action)` and `(next ?x)`.

Our system, `gdlcc`, takes as input a game description and outputs a `C++` source and header file that can be used to perform state manipulation tasks. The program works as follows. First, the game description is converted into a series of tokens and comments are removed. Second, the tokens are processed into facts and implications. During this phase, all literals, functions and relations are assigned a unique identifier. Third, a breadth-first search is performed starting from the top-level queries to determine all the sub-queries that are required. Finally, we can generate the code required to do inference. An example of how a query is translated in to `C++` is shown in Figure 1.

There are a few easy-to-implement optimizations that could improve the performance of our system. First, we can implement simple clause reordering in the implications. That is, the order in which we resolve the sub-queries in an implication rule can have a large effect on the time it takes to perform the inference. As a simple heuristic, we order the clauses by the number of unknowns that need to be resolved. The goal of this heuristic is to minimize the number of potential bindings to the unknowns that we are required to search over.

Second, some sub-queries are invoked multiple times along different query paths. Here, we are potentially duplicating the work required to perform the query every subsequent time it is required. To avoid this, we implement a simple form of *memoization*. That is, after performing a query, we remember its result in case it is required for a later query. Our system allows for memoization to be enabled or disabled, but currently there is no heuristic for deciding what should or should not be memoized. Also, we should note that any query that depends on a `(does ?role ?action)` relation cannot be memoized since this relation can change each time a successor state is generated. If we were to memoize results in this case, we may produce unusual or invalid successor states.

The code generated by `gdlcc` has a common interface, so a player program that is designed to work with said interface can be used to play any game that can be described by GDL. With this common interface, it is relatively straightforward to create a basic general game player. For the more complicated techniques that require further interactions with the game being played, further modifications to `gdlcc` may be required

Query: `(row ?x a)`
Implication: `(<= (row ?m ?x) (true (cell ?m 1 ?x)) (true (cell ?m 2 ?x)) (true (cell ?m 3 ?x)))`
Generated Code:

```
vector<tuple<1> > state::evaluate_row_01(int _a) const {
  vector<tuple<1> > results;
  vector<tuple<1> > _b = evaluate_cell_011(2, _a);
  for(vector<tuple<1> >::const_iterator _c=_b.begin();
    _c!=_b.end();
    ++_c) {
    vector<tuple<0> > _d = evaluate_cell_111((*_c)[0], 4, _a);
    for(vector<tuple<0> >::const_iterator _e=_d.begin();
      _e!=_d.end();
      ++_e) {
      vector<tuple<0> > _f = evaluate_cell_111((*_c)[0], 5, _a);
      for(vector<tuple<0> >::const_iterator _g=_f.begin();
        _g!=_f.end();
        ++_g) {
        results.push_back(tuple<1>((*_c)[0]));
      }
    }
  }
  sort(results.begin(), results.end());
  results.erase(unique(results.begin(), results.end()), results.end());
  return results;
}
```

Figure 1: Example of a query converted to C++

to extend the common interface's capabilities.

4 Results

To test the performance against `gdlcc` against a typical Prolog implementation, we implemented a simple Monte Carlo player twice, one backed by `gdlcc` and the other backed by YAP [CRACS and LIACC, 2008] with some additional code provided by Yngvi Björnsson. A Monte Carlo player randomly selects an action from the current state for each player and then performs a simulation with a random play-out policy. The reward for each player at the end of the play-out is averaged with the current score for the action that was sampled. After the move timer expires, the action with the highest average reward is selected.

The `gdlcc` player was implemented with some additional options. Specifically, it could toggle *transposition tables* [Slate and Atkin, 1977] and memoization. Transposition tables are a technique used in many game playing programs to record previously visited states usually to avoid duplicate work when searching. For example, when doing a minimax search, transposition tables can be used to record the value of a state so that if it is ever revisited it does not need to be re-expanded. Here, we used transposition tables to avoid duplicating state manipulation work. The transposition tables were not implemented in the Prolog player as the Prolog player could not record two different states in memory.

To evaluate the programs, we used four games, *Tic-Tac-Toe*, *Chess*, *Checkers* and *Connect4*. *YAP* refers to the player backed by Prolog, *gdlcc* was backed by `gdlcc` and *gdlcc tt* used `gdlcc` with transposition tables enabled. For all games with the exception of *Tic-Tac-Toe*, the `gdlcc` players had memoization enabled. A maximum of 10,000 states were stored in the transposition table for all games. We played each game twenty times and averaged the results. As all games we tested are two player games, the particular program being tested played as both players in the games. The *move clock* was five seconds on a 2.4GHz machine. For a single trial, we categorized each state visited based on how far into the play it was. That is, the first third of the states were classified as the *early* game, the second third as the *middle* game and the final third as the *end*. We recorded both the number of simulations completed and the number of states examined in the simulations; these statistics were averaged within each category. Our choice of using a Monte Carlo player, as opposed to a more sophisticated player, is due to our choice of categorization. We wanted both players to be approximately equal strength regardless of the number of simulations completed. If one player was much stronger than the other, then the comparison of the number of states examined in the different stages of the game might not be accurate.

In Table 1 we see the number of states examined per action by each player. The `gdlcc` player without transposition tables can examine between 16.5 and 18.6 times more states per action than the Prolog player on *Tic-Tac-Toe*. With the addition of transposition tables the improvement jumps to between 23.5 and 27.1 times more states per action over the Pro-

Player	Early	Middle	End
Tic-Tac-Toe			
YAP	50935	54197	52663
gdlcc	839772	896903	980867
gdlcc tt	1197720	1324552	1426093
Chess			
YAP	3240	3170	3190
gdlcc	5189	5634	5963
gdlcc tt	5091	5353	5611
Checkers			
gdlcc	9853	10159	11268
gdlcc tt	9879	9888	54312
Connect4			
gdlcc	65683	57376	53842
gdlcc tt	65683	64764	65683

Table 1: Stated examined over 5 seconds.

Player	Early	Middle	End
Tic-Tac-Toe			
YAP	7309	13212	18553
gdlcc	134932	194602	194602
gdlcc tt	164348	268363	370144
Chess			
YAP	22	31	62
gdlcc	28	41	63
gdlcc tt	28	39	58
Checkers			
gdlcc	108	217	1419
gdlcc tt	108	208	12961
Connect4			
gdlcc	3308	4612	6881
gdlcc tt	3536	5367	8156

Table 2: Simulations over 5 seconds.

log player. In *Chess* the improvement is not as pronounced, but gdlcc without transposition tables can examine between 1.6 and 1.8 times as many states as the Prolog player. We see that our transposition table implementation actually hurts our performance in *Chess*. This could be due to a few factors. First, the state description in *Chess* is much larger. Second, the states are revisited less frequently. The combination of these facts means that there is more overhead when using the transposition table coupled with less gain from their use. In *Checkers* and *Connect4* the difference that transposition tables makes is more noticeable, especially as the game moves towards the later stages. In the later stages of the game, there are fewer actions for a player to choose from and fewer moves before the game terminates. This results in many of the states visited during the simulations being stored in the transposition table. Since the state size in these two games is smaller than in *Chess*, the overhead required to access the table is relatively small compared to the amount of work that is saved by having a successful table look-up.

In Table 2 we see the number of simulations run per turn by each player follows a similar trend to the number of states examined per turn. In *Tic-Tac-Toe* we have between a 10 to 20 times speed up over the Prolog player with transposition tables. In *Chess*, the three players are quite similar across the board, again with transposition tables causing more harm than good and with gdlcc slightly edging out the Prolog player. Finally, in *Checkers* and *Connect4*, the transposition tables made quiet a difference in the end game and little to no difference towards the early game.

In summary, we see that in terms of either metric of state manipulation speed that gdlcc does no worse than Prolog and has the potential to do much better. This furthers the point that even a naive implementation of a generated state manipulation library can be beneficial in general games.

Maligne, an entrant to the 2008 General Game Playing Competition from the University of Alberta, used gdlcc for its state manipulation. It is a UCT player with a Monte Carlo simulation-based evaluation function.

5 Future Work

There is much room to improve gdlcc even further. The generated code is by no means optimal. In fact, most of its speed comes from the fact that the C++ compiler can produce very efficient machine code. When run through a profiler, we noticed that code generated by gdlcc spent an overwhelming amount of time merely allocating and freeing memory. This is likely because its use of STL containers. By inlining select sub-queries and attempting to reduce or remove the dependence on STL it would be possible to help alleviate some of this unnecessary allocation and deallocation. It is possible that the choice of when to inline might depend on information not easily accessible to the code generation program, such as what facts of a state are typically true. One possible way to combat this is to generate debugging or profiling output. This way a first draft of a player can gather and report these statistics over a few plays of the game. These statistics can then be fed back into gdlcc to make more informed decisions about how to inline.

As mentioned earlier, the order in which the sub-queries are evaluated can have a huge impact on the amount of work required to resolve the unknowns. Currently, gdlcc uses a simple heuristic to solve this problem, but a more intelligent technique could provide huge savings. Again, using profiling information could be useful in determining a good ordering.

Memoization can also be a huge win in certain games, but again it is hard to tell what should be memoized and what should be recomputed. A heuristic for choosing what to memoize or using profiling information could be of great use here. Additionally, memoization can unnecessarily use a lot of additional memory, which can cause unneeded overhead. For some games, it might actually be worse to memoize everything than to not memoize anything at all. That is, unlike clause reordering, getting memoization wrong can actually decrease performance.

As alluded to earlier, gdlcc does not provide an interface for players to do much interaction with the specifics of the game. That is, attempting to construct heuristics from a state's description is not possible with the current interface

common to all games. Additional support for these features would be useful for designing a strong player and certainly are feasible to implement.

One final issue with `gdlcc` that would be useful to address is that it can take quite some time to compile a player for some larger (in terms of description size) games. For extremely large games, `gdlcc` can produce source files that are megabytes in size. The speed at which `gdlcc` produces this code is not an issue, but the speed at which the `C++` compiler compiles the code can be. This is especially the case if optimizations are set to their highest level. If one attempts to implement some type of profiling, that would require two compilations and some training time all within a game's start clock. This additional requirement could easily exceed the start clock limits that are common currently.

6 Conclusion

We have shown that using generated code is an effective method of increasing the rate at which a general game playing program can manipulate game states. Even a naive implementation outperforms a generic Prolog package for the state manipulation tasks. Furthermore, there are many additional enhancements that can be made when generating state manipulation code that could improve performance. As searching a larger amount of the state space is likely to increase the strength of play, it is likely that players based on this method would have an advantage over those who continue to rely on a Prolog package. Unfortunately, this enhancement is merely a band-aid in the sense that it does not move us closer to achieving general intelligence. That is, the goal of winning the AAAI General Game Playing Competition does not completely coincide with the underlying research goals of the competition.

Acknowledgements

We would like to thank the past and present members of the University of Alberta's General Game Playing group for helpful conversations and feedback on `gdlcc`. Special thanks to Jonathan Schaeffer and Nathan Sturtevant for reviewing this work. Furthermore, special thanks to Yngvi Björnsson for the code used to convert a GDL game to Prolog and general game specific `C++` bindings to YAP.

References

[Banerjee and Stone, 2007] Bikramjit Banerjee and Peter Stone. General game learning using knowledge transfer. In *IJCAI*, pages 672–677, 2007.

[Banerjee *et al.*, 2006] Bikramjit Banerjee, Gregory Kuhlmann, and Peter Stone. Value function transfer for general game playing. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, 2006.

[Clune, 2007] James Clune. Heuristic evaluation functions for general game playing. In *AAAI*, pages 1134–1139. AAAI Press, 2007.

[CRACS and LIACC, 2008] CRACS and LIACC. Yap prolog, 2008. `http://www.dcc.fc.up.pt/~vsc/Yap/`.

[Finnsson and Björnsson, 2008] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*. AAAI Press, 2008.

[Finnsson, 2007] Hilmar Finnsson. Cadia-player: A general game playing agent. Master's thesis, Reykjavik University – School of Computer Science, 2007.

[Genesereth *et al.*, 2005] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI Magazine*, 26(2):62–72, 2005.

[Kaiser, 2007] David M. Kaiser. Automatic feature extraction for autonomous general game playing agents. In *AAMAS*, pages 643–649, 2007.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *In: ECML-06. Number 4212 in LNCS*, pages 282–293. Springer, 2006.

[Kuhlmann and Stone, 2007] Gregory Kuhlmann and Peter Stone. Graph-based domain mapping for transfer learning in general games. In *ECML*, pages 188–200, 2007.

[Kuhlmann *et al.*, 2006] Gregory Kuhlmann, Kurt Dresner, and Peter Stone. Automatic heuristic construction in a complete general game player. In *AAAI*, pages 1457–62, 2006.

[Love *et al.*, 2008] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical Report March 4 2008, Stanford University, 2008. Most recent version available at `http://games.stanford.edu/`.

[Schaeffer *et al.*, 1992] Jonathan Schaeffer, Joseph Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2–3):273–290, 1992.

[Schaeffer, 1983] Jonathan Schaeffer. The history heuristic. *International Computer Chess Association Journal*, 6(3):16–19, 1983.

[Schiffel and Thielscher, 2007a] Stephan Schiffel and Michael Thielscher. Automatic construction of a heuristic search function for general game playing. In *IJCAI Workshop on Nonmontonic Reasoning, Action and Change*, 2007.

[Schiffel and Thielscher, 2007b] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *AAAI*, pages 1191–1196. AAAI Press, 2007.

[Sharma *et al.*, 2008] Shiven Sharma, Ziad Kobti, and Scott D. Goodwin. Knowledge generation for improving simulations in UCT for general game playing. In *Australasian Conference on Artificial Intelligence*, volume 5360, pages 49–55. Springer, 2008.

[Slate and Atkin, 1977] D. J. Slate and L. R. Atkin. *Chess Skill in Man and Machine*, chapter 4.5 – The Northwestern University Chess Program, pages 82–118. Springer-Verlag, New York, 1977.

[Taylor *et al.*, 2007] Matthew E. Taylor, Gregory Kuhlmann, and Peter Stone. Accelerating search with transferred heuristics. In *ICAPS-07 Workshop on AI Planning and Learning*, 2007.

[Utgoff, 2001] Paul E. Utgoff. Feature construction for game playing. In *Machines that learn to play games*, pages 131–152. Nova Science Publishers, 2001.