To appear in *Proceedings of The Eighteenth European Conference on Machine Learning* (ECML-07), Warsaw, Poland, September 2007.

Graph-Based Domain Mapping for Transfer Learning in General Games

Gregory Kuhlmann and Peter Stone

Department of Computer Sciences, The University of Texas at Austin
1 University Station C0500, Austin, Texas 78712-1188
{kuhlmann,pstone}@cs.utexas.edu

Abstract. A general game player is an agent capable of taking as input a description of a game's rules in a formal language and proceeding to play without any subsequent human input. To do well, an agent should learn from experience with past games and transfer the learned knowledge to new problems. We introduce a graph-based method for identifying previously encountered games and prove its robustness formally. We then describe how the same basic approach can be used to identify similar but non-identical games. We apply this technique to automate domain mapping for value function transfer and speed up reinforcement learning on variants of previously played games. Our approach is fully implemented with empirical results in the general game playing system.

1 Introduction

We consider the problem of General Game Playing (or Meta-Game Playing as it was introduced by [10]). In this paradigm, the challenge is to design an agent that can receive descriptions of previously unseen games and play them without subsequent human input. In its lifetime, a GGP agent will encounter a variety of different games. To leverage this experience, the agent must transfer knowledge from past games in a way that is beneficial to a new task that it is given.

In the transfer learning paradigm, an agent trains on a *source* task and leverages that experience to speed up learning on a *target* task. In particular, we are interested in transferring a value function found through temporal difference learning. The intention is to provide a reasonable starting place for learning, while allowing refinement for the specifics of the target task.

Because the source task and target task may have different state and action spaces, and different goals, a prerequisite for value function transfer is a domain mapping between the two. We present a set of general mapping functions to automatically translate a value function between certain classes of game variants, a process that is typically carried out manually. Also, unlike typical transfer scenarios, an agent in our paradigm is responsible for selecting the appropriate source task from its database, rather than being given a specific source task to use. We contribute a graph-based method for recognizing similar games and prove that it is robust even when game descriptions are intentionally obfuscated.

```
1.  (role white) (role black)
2.  (init (cell a 1 b)) (init (cell a 2 b)) (init (cell a 3 b))
3.  (init (cell a 4 bk))  ...  (init (cell d 1 wr))  ...  (init (cell d 4 b))
4.  (init (control white)) (init (step 1))
5.  (<= (legal white (move wk ?u ?v ?x ?y))
6.      (true (control white)) (true (cell ?u ?v wk)) (kingmove ?u ?v ?x ?y)
7.      (true (cell ?x ?y b)) (not (restricted ?x ?y)))
8.  (<= (legal white noop) (true (control black)))
9.  (<= (next (cell ?x ?y ?p)) (does ?player (move ?p ?u ?v ?x ?y)))
10. (<= (next (step ?y)) (true (step ?x)) (succ ?x ?y))
11. (succ 1 2) (succ 2 3) (succ 3 4) (succ 4 5) ... (succ 7 8) (succ 8 9) (succ 9 10)
12. (nextcol a b) (nextcol b c) (nextcol c d)
13. (<= (goal white 100) checkmate)
14. (<= (goal black 100) (not checkmate))
15. (<= terminal (true (step 10)))
16. (<= terminal stuck)
```

Fig. 1. Partial game description for "Minichess". GDL keywords shown in **bold**.

The following section provides background on GGP and discusses the key elements of the game description language as well as our method for analyzing such descriptions. Section 3 introduces our graph-based game recognition algorithm, sketches a proof of its robustness, and outlines the recognized game variant classes. In Section 4, we discuss how to map learned value functions between game variants. Our complete approach is evaluated in the GGP framework in Section 5. Section 6 surveys related work and Section 7 concludes.

2 General Game Playing

The general game playing scenario adopted for this work is taken from the AAAI General Game Playing competition [8]. In the competition setup, the *Game Manager* connects to each player process and sends the game description along with time limits called the *start clock* and *play clock*. Players have the duration of the start clock to analyze the game description before the game begins and the duration of the play clock to choose their moves each turn. The game continues until a terminal state is reached. No human intervention is permitted at any point: the general game player must be a complete and fully autonomous agent.

2.1 Game Description Language

For an agent to interpret a game, it must be described in a well-defined language. In the Game Description Language (GDL), used in the competition, games are modeled as state machines. An agent can derive its legal moves, the next state given the moves of all players, and whether or not it has won by applying resolution theorem proving. Part of the description for a game called "Minichess" is shown in Figure 1. A GGP agent must be able to play any game, given such a description. We illustrate the features of GDL through this example.

First, GDL declares the game's roles (line 1). "Minichess" has two roles, `white` and `black`. Next, the initial state of the game is defined (2–4). Each functional term inside an `init` relation is true in the initial state. Besides `init`, none of the tokens in these lines are GDL keywords. The predicates `cell`, `control` and `step` are all game-specific. With the exception of goal values, even the numbers do not have any external meaning. If any of these tokens were to be replaced by a different token throughout, the meaning would not change.

GDL also defines the set of legal moves available to each role through `legal` rules (5–8). The `<=` symbol is the reverse implication operator. Tokens beginning with a question mark are variables. The `true` relation is affirmative if its argument can be satisfied in the current state. The state transition function is defined using the `next` keyword (9–10). The `does` predicate is true if the given player selected the given action in the current state. Finally, GDL defines rules to determine when the game state is `terminal` (15–16). When the game ends, each player receives the reward defined by the game's `goal` rules.

A game description may define additional relations to simplify the conditions of other rules and support numerical relationships. For instance, the `succ` relation (11) defines how the game's step counter is incremented, and the `nextcol` relation (12) orders the columns of the chess board. Identifying such relationships is valuable because they bridge logical and numerical representations.

2.2 Automated Domain Analysis

An approach common to most computer game playing systems is heuristic search, in which game states are evaluated based on the contribution of various features. Although these features are typically supplied by the system designer [3, 11], the general game playing setting prohibits such human involvement, motivating the development of automated methods.

In prior work on automated domain analysis [9], we developed techniques for generating features automatically from GDL game descriptions. The structures identified during domain analysis are also valuable in domain mapping. One of the most basic structures to look for is a *successor* relation. This type of relation induces an ordering over a set of objects. We have already seen two examples in lines 11–12 of Figure 1. A major challenge for automated domain analysis in GGP is that, during the competition, the description's non-keyword tokens are scrambled to prevent the use of lexical clues. In a competition setting, the same successor relations may look something like: `(tcsh pico cpio) (tcsh cpio grep) ... (tcsh echo ping)`.

Our system can still identify these relations as successors because order is a structural, rather than lexical, property. Based on these relations, the agent identifies additional structures such as a step counter, which is a functional term in the game's state that increments each time step. Our system identifies it by looking for a rule such as the one on line 10 of Figure 1.

Another example of a structure that our agent attempts to identify is a *board*. A board is a two dimensional grid of cells that change state, such as a chess or checkers board. Once a board is identified, the system looks for *markers*, which are objects that occupy the cells of the board and *pieces*, special markers that occupy only one cell at a time. In "Minichess", the white rook, `wr`, and the black king `bk` are examples of pieces. Games like Othello have only markers.

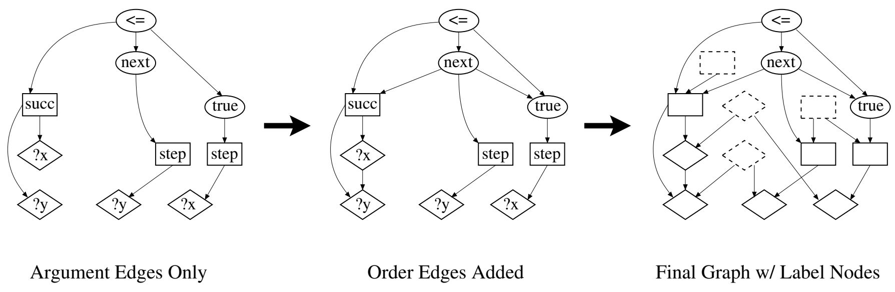


Fig. 2. Rule graph construction for step counter rule on line 10 of Figure 1.

3 Graph-Based Domain Mapping

An important step in transferring knowledge from a known game to a new, unknown game is recognizing the extent to which the games are similar. Because their rules are reordered and tokens scrambled during a competition, it is not clear from the GDL descriptions even if two games are identical. To address this problem, we introduce the concept of rule graphs, canonical form graph structures that capture the important aspects of the game description while ignoring inconsequential elements such as token names and rule order.

3.1 Rule Graphs

Rule graphs are colored, directed graph representations of game rules. To describe rule graph construction, we use the example of the step counter rule in line 10 of Figure 1. Conceptually, a rule graph is constructed in three stages. These stages for the step counter graph are show from left to right in Figure 2.

First, the graph contains a node for each logical sentence, relational sentence, constant and variable in the rule. For each of these terms, an *argument edge* is added from the term's node to all of the term's arguments. Variable nodes are shown as diamonds. Logical sentence nodes are labeled by the sentence's operator. Relational sentences and constants are labeled with their functor's name and shown as circles, if they are keywords, and shown as squares, otherwise. The resulting graph is shown on the far left in Figure 2. For the purpose of matching, each circular node with a different label is considered a different color.

As shown in the center graph in the figure, we add additional edges to force ordering constraints on arguments. For each relational sentence with arity greater than one, we add an *order edge* from each argument to the argument that follows it. The edge between the two arguments of the `succ` relation in the figure is an example of such an edge. Also, we constrain the consequent of an implication to precede the antecedents without enforcing an order on the antecedents themselves. We achieve this constraint by adding edges from the consequent to each of the antecedents, as in the edge from `next` to `succ` and from `next` to `true`.

Finally, we must identify which variables and constants are the same without keeping the specific labels. For each unique variable in a rule and for each unique non-keyword constant in the game description, we create a new *label node* and add edges to each of their instances. Variable label nodes are shown as dashed diamonds and non-keyword constant label nodes as dashed squares. The final graph for the step counter rule is shown on the far right in Figure 2.

To determine if two rules graphs are isomorphic, one must simply use any off-the-shelf graph isomorphism algorithm. Through informal experimentation we found VF [4] to be relatively efficient for the structure of rule graphs.

3.2 Correctness Proof

In this section, we sketch a proof for the correctness of rule graph isomorphism as a means to determine if two games are the same, modulo scrambling. We begin with a few definitions, followed by the formal statement of the theorem.

In GDL, an *atomic sentence* is a relation constant of arity k applied to k terms: $p(x_1, \ldots, x_k)$ or equivalently $p(x_1^k)$. An example of an atomic sentence is father(bob, X). If k is 0 then the sentence is called an *object constant.* A *term* is either a *variable* (e.g. X) or an atomic sentence. A more complex example of an atomic sentence is the following: f(X, g(Y, h(p)), q). The constants may be user-defined, such as `cell`, or GDL keywords such as `not`, `terminal`, or `distinct`.

A *rule* is an implication of the form: $h \Leftarrow b_1 \wedge \cdots \wedge b_n$, where h, the head, and each b_i in the body are atomic sentences. Although GDL supports disjunction in the body, it is always possible to remove this disjunction and write rules in this form. Because conjunction is associative and commutative, we can represent the body of a rule as a set, B. Therefore, we represent such a rule as a pair $h \Leftarrow B$ where $B = \{b_1, b_2, \ldots, b_n\}$. If the set B is empty, then the head of the rule is an unconditionally true fact.

A *game description* is a set of rules. A *variable scrambling* is a one-to-one function over the variable labels present in a rule. A *constant scrambling* is a one-to-one function over the constant labels present in a game description. A *game scrambling* of a game description γ is a constant scrambling of γ and a set of variable scramblings, one for each rule in γ. Two game descriptions γ and γ' are *scramble-equivalent* if there exists a scrambling η such that $\gamma' = \eta(\gamma)$.

Theorem *Two game descriptions γ and γ' are scramble-equivalent if and only if the rule graph G created from γ and rule graph G' created from γ' are isomorphic.*

Proof Sketch.
The forward implication of the theorem is fairly straightforward to prove. If two game descriptions are scramble-equivalent, then their corresponding rule graphs will be isomorphic. A simple argument for this statement is that the graph construction algorithm is deterministic and does not depend on the exact names of the non-keyword tokens.

The reverse direction is a bit more subtle. We will prove that isomorphic rule graphs imply scramble-equivalent game descriptions by induction on the size of the game description. Beginning with the base case of $\gamma = \emptyset$, we can construct any game description by composing the following operations:

1. Add new rule with object constant head and empty body:
   $\gamma \longrightarrow \gamma \cup \{c \Leftarrow \emptyset\}$
2. Add object constant as antecedent of existing rule:
   $\gamma \cup \{h \Leftarrow B\} \longrightarrow \gamma \cup \{h \Leftarrow B \cup \{c\}\}$
3. Append object constant to some atomic sentence in head of existing rule:
   $\gamma \cup \{p(\ldots r(x_1^k)\ldots) \Leftarrow B\} \longrightarrow \gamma \cup \{p(\ldots r(x_1^k, c)\ldots) \Leftarrow B\}$
4. Append variable to some atomic sentence in head of existing rule:
   $\gamma \cup \{p(\ldots r(x_1^k)\ldots) \Leftarrow B\} \longrightarrow \gamma \cup \{p(\ldots r(x_1^k, X)\ldots) \Leftarrow B\}$
5. Append object constant to some atomic sentence in body of existing rule:
   $\gamma \cup \{h \Leftarrow B \cup \{p(\ldots r(x_1^k)\ldots)\}\} \longrightarrow \gamma \cup \{h \Leftarrow B \cup \{p(\ldots r(x_1^k, c)\ldots)\}\}$
6. Append variable to some atomic sentence in body of existing rule:
   $\gamma \cup \{h \Leftarrow B \cup \{p(\ldots r(x_1^k)\ldots)\}\} \longrightarrow \gamma \cup \{h \Leftarrow B \cup \{p(\ldots r(x_1^k, X)\ldots)\}\}$

For each of the above operations, we construct a corresponding abstract rule graph, G. This graph can be divided into two partitions: the nodes and edges present prior to applying the operation, and those added by the operation. By the definition of isomorphism, the isomorphic rule graph, G', can then be partitioned in the same way. By applying the graph building algorithm in reverse, we get a partitioned game description γ'. By the induction hypothesis, the original subgraph isomorphism implies scramble-equivalent subgames. What remains is to show that there exists a scrambling compatible with the subgame scrambling that makes the induction step rule equivalent to its partner in the isomorphic game. The same procedure proves the induction step for each operation.

3.3 Identifying Similar Games

While it is undoubtedly useful to recognize identical games, the applicability of our algorithm is much greater if we extend it to similar, but non-identical games. Our approach is to continue using identical rule graph isomorphism, but to test against generated variants of previously played games. We begin by identifying the classes of variants that we have determined to produce small, local changes to rule graph structure. Each variant defines a transformation procedure, which modifies the original rule graph by adding and/or deleting nodes and edges.

The first class of variants are those that change only the initial state of the game. By comparing all of the rules other than the initial state declarations, the standard graph isomorphism algorithm can identify these variants: **Num Markers**, in which the number of markers on the board differs and **Piece Configuration**, in which the location of pieces is different.

A more challenging class of variants to identify are those that change one or more of the game's successor relations. For example, the `nextcol` relation defined on line (12) of Figure 1 could be made longer by adding another rule: `(nextcol d e)` or shorter by removing the rule `(nextcol c d)`. Alternatively, we could make the sequence cyclic by adding the rule: `(nextcol d a)`. Each of these game description changes correspond to rule graph transformations. By applying these candidate transformations prior to matching, we can identify the **Board Size** variant, in which the length successor relation ordering the coordinates of one or more of the board dimensions has been changed. The **Cylindrical/Toroidal Board** variant makes cyclic the successor relation that orders one or more of the coordinates of the board. Finally, **Step Limit** alters the maximum number of steps before forced termination by changing the step number in the termination condition and expanding the counter's successor relation as necessary.

Step Limit is a composite variant in that it modifies both a successor relation and a goal condition. Another goal variant is **Inverted Goal**, in which the constants “100” and “0” are swapped in the second argument of all instances of `goal`. In **Switched Role**, the rule graph is unchanged but the player’s assigned role is different (e.g. playing as O instead of X in tictactoe). Lastly, in the **Missing Feature** variants, a state feature present in the source task state is absent in the target task state. The transformation procedure removes all instances of the feature and the rules that include it, (e.g. removal of a step counter).

Offline, the agent generates a rule graph for each applicable transformation of each previously played game. When faced with a new game, the agent generates its rule graph and attempts to match it against every graph in the database. Non-matching graphs are typically rejected very quickly; only correct matches require any significant amount of computation. With a database of roughly 100 games, the entire process never requires more than 27 secs on a 2.80GHz machine.

Although this approach can detect quite complicated transformations of games, there are limits to its power. Game variants that affect many rules at once are particularly difficult to handle. For example, the board topologies in 3 and 4 player Chinese checkers games differ by too many rules to describe their difference as a concise, generally-applicable transformation procedure.

To this point, we have described our procedure for identifying game variants. In the next section, we describe our approach to transferring knowledge between these games for the purpose of speeding up learning.

4 Value Function Transfer

The approach detailed in this work transfers a learned value function from a source task to initialize the value function of a target task, identified to be similar through the graph-based method described in the previous section. Before introducing our approach to value function transfer, we provide some background on the reinforcement learning paradigm and detail the assumptions and design choices of our learning algorithm.

4.1 Reinforcement Learning in Games

In a Reinforcement Learning (RL) problem [13], an environment is modeled as a Markov Decision Process (MDP), defined by a transition function, T, and a reward function, R. Many common algorithms for solving RL problems are based on learning a value function, Q, which approximates the expected long-term reward for executing action a in state s.

In GGP, R is known, but for multiplayer games, T is only partially known, because the transition function depends on the opponent’s unknown policy. If we consider only turn-taking games, in which the next state is uniquely determined by the agent’s action on its turn, then we need only learn a function V over what are commonly called *afterstates*. Although it is still possible to learn Q, V has fewer values, increasing generalization. Also, this representation simplifies transfer mapping by requiring only a mapping between the states of the two tasks and not the actions.

Mapping	**Initial $V_T[s]$**	**Applicable Variants**
Direct	$V_S[s]$	Step Limit Num Markers Piece Configuration
Inverse	$100 - V_S[s]$	Inverted Goal Switched Role
Average	$1\frac{\vert}{B}(s)\vert \sum_{s' \in B(s)} V_S[s']$	Board Size Missing Feature

Fig. 3. Value function initialization formulas for three mappings, along with applicable game variants. V_S and V_T are the value functions for the source and target tasks, respectively. $B(s)$ is the set of source task afterstates mapped to target afterstate s.

A popular algorithm for learning value functions is an incarnation of temporal difference learning called Sarsa [13]. Taking into account the assumptions discussed thus far and that R is defined only for terminal states, our learning algorithm can be described by the following update rule:

$$V[s_{t-1}] \leftarrow V[s_{t-1}] + \alpha \cdot \begin{cases} (R(s_t) - V[s_{t-1}]) & \text{if } s_t \text{ is terminal,} \\ (V[T(s_t, a_t)] - V[s_{t-1}]) & \text{otherwise.} \end{cases}$$

where α is the learning rate (0.3 in our experiments), s_t is the current state, a_t is the agent's chosen action, and s_{t-1} is the afterstate following the agent's previous action. This algorithm and the rest of the transfer learning approach described in this work make no assumptions about the representation of V. In our experiments, each $V[s]$ is stored as a single real value in a table. However, there is nothing in principle that would prevent our method from working with a function approximator, such as a neural network.

4.2 Value Function Mapping

To translate afterstate values in the source into afterstate values in the target we must construct a mapping between their state spaces. The appropriate mapping depends on the identified relationship of the tasks. In Figure 3 we propose three possible mapping functions and, for each, identify applicable game variants.

The *direct* mapping simply copies the value of an afterstate in the source task directly to afterstate value in the target task. This mapping assumes that the two tasks have the same state space and roughly the same goal condition. Because the Step Limit variant effects only the duration of the game and Num Markers and Piece Configuration effect only the game initial state, the direct mapping seems to be appropriate.

The *inverse* mapping also assumes that the state space is the same between tasks, but that the goal has changed. In particular, it assumes that the goal of the target task is the exact opposite of the source task. This mapping is clearly applicable in the case of Inverted Goal. Assuming that the game is zero sum, then the mapping is also appropriate for Switched Role.

The final mapping, *average*, assumes only that there is a function, B, that for a given target afterstate, returns the set of relevant source afterstates. In the case of Missing Feature, $B(s)$ would return all source task afterstates with feature values matching those of s for the features remaining in the target afterstate. The Board Size variant uses a particular B, detailed in the next section.

4.3 Case Studies

In this section, we discuss the application of the complete mapping procedure to three concrete examples, one from each mapping category. Our first transfer scenario involves a miniature checkers game played on a 5×5 board. The rules are identical to normal checkers, except that an available jump must be taken.

This source-target pair is an example of the Num Markers variant. In the target task, each player begins with only 4 pieces instead of 5. The goal in both tasks is the same: to capture your opponent's markers before they capture yours. It is reasonable to assume that there will be some overlap in the states visited in the source task and those in the target task, so the Direct mapping appears to be the logical choice. At the same time, the degree to which the transfer may help (or hurt) must be answered empirically.

In the class of games appropriate for the Inverse mapping, we looked at the game of tictactoe and a variant in which the goal is inverted. Because the goal of the game is the opposite of the original goal, highly valued states of the source task should have low values in the target task and vice-versa.

Finally, "Minichess", the chess variant introduced in Section 2.1 is used as an example of a Board Size variant. For a target task with board size $N \times N$, we assume that the source task has a board size of $N - 1 \times N - 1$. In particular, we use the 3×3 game as the source task and the 4×4 game as the target task. The afterstate mapping function B, is constructed as follows. An afterstate represented by the 4×4 board in the target task is mapped to four candidate afterstates in the source task, each represented by a 3×3 board, which we call a *subboard*. The first subboard consists of the top-left 3×3 cells of the board. The next subboard consists of the top-right cells, continuing clockwise. Figure 4 shows a source task afterstate and its four candidate subboards. Each of these candidates is not necessarily a valid afterstate in the source task. If no subboards are valid, the afterstate value is initialized to the default value.

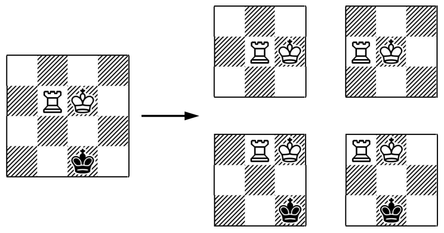
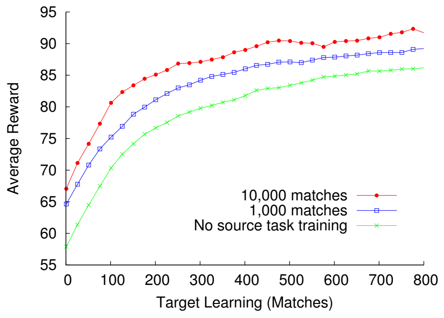


Fig. 4. Full 4×4 "Minichess" state and all four candidate subboards.

5 Experiments

We conducted experiments to determine the impact of value function transfer in each of the described scenarios, using different amounts of training on the



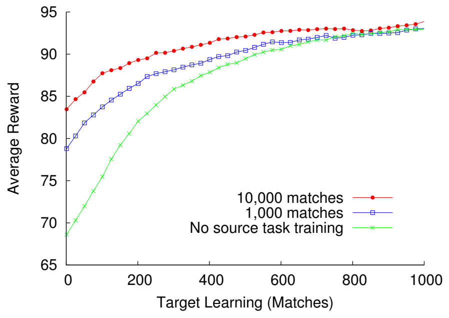


Fig. 5. Transfer learning results with varying source experience. *Left:* "Checkers" game with Num Markers variant. *Right:* "TicTacToe" game with Inverted Goal variant.

source task. Each learning trial consists of an independent source task learning phase followed by a target task learning phase. The target task agent's value function is initialized by the source task value function, according to the domain's generated mapping function. Each trial was repeated 30 times, with a sliding window average of 100 matches used to generate learning curves. To determine statistical significance, we evaluated the curves at several points using a one-tailed T-test with 95% confidence. All figures show averaged learning curves.

The results of our "Checkers" experiments are shown on the left in Figure 5. Value function transfer produces an initial performance improvement that persists until convergence. The other two curves eventually catch up, beyond the scale of the graph. The visible differences are statistically significant for all data points shown. The average number of state values transferred for the most experience player was 1,113, which is roughly a third of all unique afterstates encountered during target task learning. To make learning more valuable in these experiments, we used an opponent agent trained for an extensive period of time with temporal different learning rather than a random player.

The "TicTacToe" results, shown on the right in Figure 5, again demonstrate the positive impact of transfer. In this case, the average hit rate was 68% for the learner with 10,000 matches of source task experience. This substantial reuse of source task values helps to explain the significance of the transfer benefit. The performance improvement of the player with 10,000 matches of source task experience is significantly better than the from-scratch learner until 700 matches.

To make the problem more interesting for learning, at the start of each match, for both source and target learning, the state is initialized to a random configuration of the pieces. In only about half of the initial states can the white player force a win. However, in our experiments, the black player is controlled by a one-move lookahead player, and thus, by exploiting the suboptimality of the opponent, a learner is able to earn an average score above 50.

The transfer results for "Minichess" are shown in Figure 6. Transfer clearly improves the learner's initial performance. The experienced learners do significantly better than the agent learning from scratch up until 1,200 target matches. The agent learning from scratch then performs better than the experienced players between 2,500 and 4,000 matches. The fact that the slowdown is more pronounced for the 1,000 match learner than for the 100 match learner indicates that the agent may be overfitting to the source task. The negative effect is short-lived, however, and by 4,000 target task matches, all three learners converge.

6 Related Work

Graph theoretic structures have long been recognized for their value to logic programming. Scheffer et al. [12] define an *occurrence graph* and demonstrate its utility in efficiently solving the θ-subsumption problem for ILP. This graph relates the shared variables between a pair of clauses, but does not relate symbol names across clauses like our rule graphs.

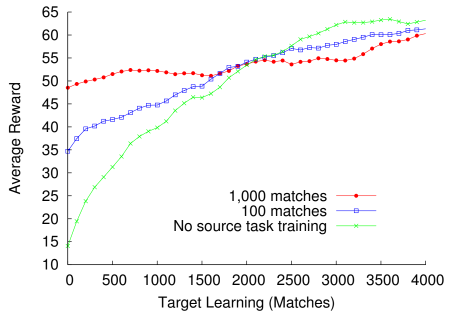


Fig. 6. Transfer from 3×3 board to 4×4 board in "Minichess".

In other work [6, 5], *dependency graphs*, used to check for consistency in logic programs with negation, have been extended to apply under the stable model semantics. Dependency graphs are defined over the predicate symbols of a knowledge base and capture a different abstraction than that in our work.

The most similar graph structure to our own is that defined by Xu and Tao [15]. With this structure, they demonstrate that the isomorphism problem for definite logic programs (those with only Horn clauses) is equivalent to the graph isomorphism problem. Our proof in Section 3.2 extends this result to general Datalog programs with negation.

In their recent work, Taylor et al. [14] demonstrate that value function transfer is able to speed up learning between tasks when the domain mapping is given. Their work also makes progress towards automating mapping by employing a classification algorithm to map actions between tasks. This method requires that the states are defined in terms of objects and each state feature is associated with one of those objects.

Other work on Relational Reinforcement Learning (RRL) has shown that by maintaining the relational structure of the domain in the representation of the value function, it is possible to learn to solve differently parametrized tasks simultaneously [7]. As our work makes no assumptions about value function representation, future work may reveal RRL to be complementary to our approach. RRL has even been applied in GGP. Asgharbeygi et al. [1] learn the values of handcrafted relational predicates to speed up learning considerably.

Another successful example of transfer learning in GGP is the work of Banerjee and Stone [2], in which features of the game tree alone are leveraged for transfer. Although the features are somewhat expensive to compute because they require search, the learner is able to transfer learned knowledge across games with significantly different state and action spaces.

7 Conclusion

This work makes progress toward the complete automation of domain mapping for value function transfer learning. The first main contribution is the *rule graph* structure, which is useful for representing games in canonical form. Beyond its use in this paper, the rule graph is likely to be of general interest to the GGP community, as a way to leverage past experience. The second main contribution of this paper is that rule graphs, along with a set of identified variant classes, can be used as a practical method for recognizing variants of previously played games and speeding up learning in the General Game Playing framework.

References

1. N. Asgharbeygi, D. Stracuzzi, and P. Langley. Relational temporal difference learning. In *ICML*, 2006.
2. Bikramjit Banerjee and Peter Stone. General game learning using knowledge transfer. In *The 20th International Joint Conference on Artificial Intelligence*, pages 672–677, January 2007.
3. Murray Campbell, A. Joseph Hoane Jr., and Feng Hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.
4. L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. An improved algorithm for matching large graphs. In *Proc. of the 3rd IAPR-TC-15 Internation Workshop on Graph-based Representations*, pages 149–159, Italy, 2001.
5. Stefania Costantini. Comparing different graph representations of logic programs under the answer set semantics. In *Proceedings of the AAAI Spring Symposium on Answer Set Programming*, 2001.
6. Yannis Dimopoulos and Alberto Torres. Graph theoretical structures in logic programs and default theories. *Theoretical Computer Science*, 170(1):209–244, 1996.
7. S. Dzeroski, L. De Raedt, and K. Driessens. Relational reinforcement learning. *Machine Learning*, 43:7–52, 2001.
8. Michael Genesereth and Nathaniel Love. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2), 2005.
9. Gregory Kuhlmann, Kurt Dresner, and Peter Stone. Automatic heuristic contruction in a complete general game player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, July 2006.
10. Barney Pell. Strategy generation and evaluation for meta-game playing. PhD thesis, University of Cambridge, 1993.
11. Jonathan Schaeffer, Joseph C. Culberson, Norman Treloar, Brent Knight, Paul Lu, and Duane Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53(2-3):273–289, 1992.
12. Tobias Scheffer, Ralf Herbrich, and Fritz Wysotzki. Efficient theta-subsumption based on graph algorithms. In *Inductive Logic Programming Workshop*, 1996.
13. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Cambridge, MA, 1998.
14. Matthew E. Taylor, Shimon Whiteson, and Peter Stone. Transfer via inter-task mappings in policy search reinforcement learning. In *The Sixth International Joint Conference on Autonomous Agents and Multiagent Systems*, May 2007.
15. Dao-Yun Xu and Zhi-Hong Tao. Complexities of homomorphism and isomorphism for definite logic programs. *Journal of Computer Science and Technology*, 20(6):758–762, November 2005.