Distance-Based Mapping for General Game Playing

Joshua D. A. Jung and Jesse Hoey
Cheriton School of Computer Science
University of Waterloo
{j35jung,jhoey}@uwaterloo.ca

***Abstract*—In the field of General Game Playing (GGP), artificial agents (bots) may be required to play never-before-seen games with less than one minute to initialize and train. Although tabula rasa approaches, like Monte-Carlo Tree Search, are popular in this domain, they do not leverage information from the many different games that a bot has previously encountered. A major barrier to transfer learning has been the difficulty in identifying similar features in the rule descriptions of two different games. We present two methods, called MMap and LMap, for heuristically approximating a distance between two games' graphs, and producing a mapping for the symbols of one to the other, thereby enabling transfer. We evaluate the effectiveness of these methods across a variety of transfer scenarios, and find that both methods are far more accurate than a simpler baseline mapper. MMap is found to be more robust than LMap, but LMap is much faster, and so more suitable for general use in GGP.**

I. Introduction

In the enduring competition between human and machine, recent years have been unkind to humanity. We have seen top-level human players lose to artificial agents (bots) at games like Chess, Go, and even video games like Dota and StarCraft. Each of these victories for machine-kind has marked a substantial milestone in artificial intelligence research. However, these bots are typically extremely specialized, requiring many human-hours to design and many thousands (or more) of machine-hours to train. Among these bots, AlphaZero stands out as a generalist, but even AlphaZero can require days or weeks to train on a specific problem [1].

General Game Playing (GGP) is a subfield of artificial intelligence that demands bots be both general and fast. A bot may be given a set of rules for a game that it has never seen before, and expected to begin playing that game less than one minute later. Although headway has been made in adapting deep learning (and specifically, AlphaZero) to GGP, training time is still on the order of hours [2]. For now, approaches like Monte-Carlo Tree Search (MCTS) remain popular [3], as they are able to learn quickly without prior knowledge. However, this tabula rasa approach to learning leaves useful information on the table. A bot may encounter many games (as in the Lifelong Learning paradigm), and learn to play each of them quite well. Should it come across a game that is similar to one it has already seen, it should be possible to gain an advantage by leveraging that prior knowledge through transfer learning.

Transfer learning has been popularized by deep learning, but is not limited to deep neural networks. Other techniques, like policy or value transfer, have been used successfully for games with many shared features. However, part of the foundation of GGP research is deliberately making it difficult to recognize the components of a game, via the process of obfuscation. During obfuscation, every word in a game's description that is not a base part of the Game Description Language (GDL), is replaced by a different, arbitrarily chosen word. In Chess, for example, every occurrence of the word 'knight' might be replaced by 'firetruck'. This prevents word choice from influencing a bot's behaviour, but leaves the structure of the rules intact. This structure can be efficiently cast into a graph representation, but finding similar features among graphs is a costly process. As a result, transfer learning has seen only limited use in GGP.

In this paper, we present novel methods for forming a mapping from the elements of a source game (which we will call G_1) to a target game (G_2) with the goal of enabling a GGP bot to cast information from a newly encountered game to one that it has seen before. From there, any manner of transfer learning may be applied.

Our method is the first to approximate an edit distance between nodes in two different rule graphs. This not only allows symbols to be mapped from G_1 to G_2, but also produces an overall distance that can be used to judge the quality of a mapping, and choose the best game from which to transfer. In general, finding the edit distance between two graphs is a well-known NP-Hard problem, so it is necessary to employ novel heuristics and greedy strategies to remain as lightweight as possible. The faster a mapping algorithm can be made, the more time a bot will have for self-play during its short initialization period.

The main contributions of this paper are:

1) A heuristic rule graph search that approximates the similarity of nodes, while limiting the number of other nodes expanded;
2) Two greedy methods (called MMap and LMap) for producing a symbol mapping from G_1 to G_2 by approximating their edit distance;
3) An evaluation of the effectiveness of these methods across a variety of transfer scenarios, which show MMap to be more robust, LMap to be faster, and both to be much more accurate than a simpler baseline mapper.

II. Background and Related Work

A. General Game Playing

General Game Playing began as a way to promote the research of generally capable, rather than purpose-specific,

978-1-6654-3886-5/21/$31.00 ©2021 IEEE

```
1: (<= (goal black 100)
2:     (true (piece_count red ?rc))
3:     (true (piece_count black ?bc))
4:     (greater ?bc ?rc))
```

Fig. 1. An example GDL statement from Checkers, in the GGP base game repository. It specifies that if the game ends and black has more pieces than red, then black wins (i.e. receives a score of 100).

artificial agents. Much of early GGP research was centered around the annual GGP competition, which was first held in 2005 [4]. The competition standardized the setting in which bots played games. Competing bots were given the rules of the game and allowed a period of time for initialization (typically on the order of 90 seconds), and then a shorter period of time (typically on the order of 10s of seconds) to play each turn.

Game rules were written in Game Description Language, which is the de facto language of GGP. GDL is a logic programming language (like Prolog) with a small number of built-in relations that are particular to playing games [5]. The game state is determined by the set of facts that are true at a given time, and game dynamics are given by a set of rules. Any words that are not part of the built-in vocabulary of GDL are obfuscated (i.e. replaced by arbitrary, unrelated words) before the game description is sent to a bot, thereby denying the use of semantic information through the particular vocabulary used. Figure 1 gives an example of (non-obfuscated) GDL code from the description of Checkers [6].

B. Rule Graphs

Stanford's GGP framework [7] includes code for parsing each GDL statement into a directed tree. This tree may be further refined into a rule graph[1], using a method like those described by Genesereth [6] or Kuhlmann [8]. This produces a coloured, directed graph, where the colours capture built-ins (i.e. logical connectives and GDL keywords), and assign types to user-defined entities (e.g. proposition, variable). The names of these user-defined entities (which may be obfuscated) are discarded. For each user-defined entity, there is an *occurrence* node for each time that entity appears in the GDL code, and a single *symbol* node that links all occurrences together. If the entity takes N arguments, there will also be N *argument* nodes connected to the symbol node[2]. Edges (with few exceptions) represent parent-child relationships in the GDL (e.g. the children of an AND node are its arguments). An example of a rule graph is given by Figure 2.

The size of a rule graph varies with the complexity of the game's description. This is distinct from the actual difficulty of a game (i.e. the size of its state space), though the two tend to be correlated. Table I gives the number of nodes generated for an assortment of games familiar to humans. The largest of these (Chess) contains over 5,000 nodes.

Although a rule graph may not be a minimal representation of a game, Kuhlmann and Stone [9] proved that if two rule graphs are isomorphic, then the games that they represent are equivalent. Jiang et al. [10] also developed methods for proving game equivalence. If a bot has previously played a game that is equivalent to its current task, then it is extremely desirable to identify that game, because all previous knowledge can be immediately transferred. However, this approach is limited to games that are exactly the same. Kuhlmann and Stone [9] extended it by looking for isomorphisms within a set of predetermined variations. For example, they were able to identify games that were identical, but for different board sizes, numbers of pieces, or turn limits. We would like to be more general than this by identifying similarities without needing to predefine patterns for which to search.

[1]Rule graphs are not the only useful structures that can be obtained from GDL. Propositional networks [6] are commonly used in the GGP literature.

[2]This applies only to Genesereth's version.

C. Other Methods for Mapping and Transfer

Others have done transfer between non-identical games by extending the Structure Mapping Engine [11]. Klenk and Forbus [12], were able to establish mappings between kinematics problems, while Hinrichs and Forbus [13] mapped between games where a character moves on a 2D grid. Both showed positive transfer in their respective domains. These works are similar in purpose to our own, though execution times do not seem to have been a major concern, as they were not reported.

Although not strictly a kind of mapping, feature extraction is a common practice in GGP that allows a bot to find information known to be useful in a variety of games. Since board games are encountered frequently, there are methods for finding sequences of numbers, board coordinates [14], [15], and moveable pieces [16], which can then be combined to produce heuristics. Game-independent features may also be discovered autonomously [17], and whole games may be decomposed if they are made up of smaller sub-games [18].

III. Method

A. Rule Graph Generation

We begin with a rule graph for a known game that will act as G_2 (or potentially, many stored rule graphs, from which we will choose the best), and we must process a new GDL description into a rule graph to act as G_1. We do so using the method described by Genesereth [6] because, unlike Kuhlmann's method [8], it does not require coloured edges, which makes processing the graph somewhat simpler. A visual representation of a rule graph is given by Figure 2 for the GDL statement in Figure 1. We defer a full description of the graph generation process to Genesereth's work, but we would like to draw attention to a few of the node types (colours) that will be important in the descriptions to follow.

In Figure 2, *symbol* nodes are given a dashed border. There is one symbol node for each unique user-defined symbol in a GDL description, positioned such that all of the relationships of that symbol can be discovered by searching from its symbol node. Of these nodes, some represent variables (purple), and others represent non-variables (white). Variables are important to the logic of a game, but cannot appear in the game state, which must be fully instantiated. It is therefore the non-variable symbol nodes that we are interested in mapping.

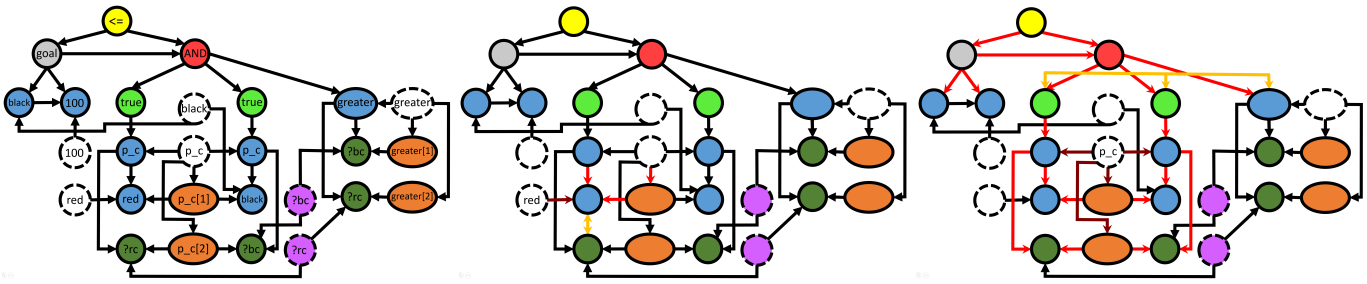

Fig. 2. (Left) The rule graph generated from the GDL code in Figure 1. Names are overlaid for clarity, although they are not normally visible. Symbol nodes have dashed edges (variable symbols are purple, non-variables are white). Non-variable occurrence nodes are blue. Variable occurrences are dark green. Argument nodes are orange. Each built-in GDL keyword has a unique colour. (Middle) Coloured, barbed arrows indicate the edges that can be discovered by our distance algorithm, starting from the 'red' symbol node. Dark red edges are part of the initial expansion, lighter red arrows are part of subsequent expansions, and yellow double-headed arrows are sibling relationships that are not true edges in the graph. (Right) Edges discovered, starting from 'piece_count' (p_c).

Each symbol will have an outgoing directed edge to some number of *occurrence* nodes, its children. An occurrence node (blue for non-variables, dark green for variables) represents one place that the corresponding symbol appears in the GDL game description. If it is passed arguments, then it will have their occurrence nodes as its children, and if it is itself an argument to some other occurrence, that occurrence node will be its parent. If a symbol takes N arguments, then its node will additionally have N *argument* nodes (orange) as children. These nodes are connected to every occurrence that appears in a particular argument position.

Importantly, the only nodes that are able to form connections from one GDL statement to another are symbol and argument nodes. All other types of nodes form connections that are local within the original GDL. In order to keep the execution time of our graph searches manageable, it is therefore necessary to be careful in the handling of symbol and argument nodes. Hereafter, if we do not specify the type of a symbol node, we are referring to a non-variable.

B. Approximate Edit Distance

In order to find a mapping from the symbols of G_1 to those of G_2, we need a way to measure the similarity of nodes (n_1 and n_2) in different graphs, using only colour and graph structure. To that end, we approximate a kind of edit distance, which requires counting the number of changes needed to change the part of G_1 reachable from n_1 into the part of G_2 reachable from n_2 (or vice versa, as the process is symmetric). We are allowed to add a node, delete a node, or change the colour of a node, at a cost of 1 unit distance per operation. Our notion of edit distance differs from common usage in two important ways. First, if the colour of two nodes is different, we immediately truncate the search and assign a distance of 1. Second, if their colours match, then distance is given by:

$$\frac{\sum_{neighbours} distance}{(\#\ of\ neighbours) + 1} \tag{1}$$

where neighbours are generally child, parent, and sibling nodes, and distance is (an approximation of) the minimum distance from pairing all neighbours of n_1 to neighbours of n_2. So, if n_1 and n_2 are of different colours, then their distance is 1, which is the highest possible value. If they are the same colour, and their children (and all of their successors) have matching colours, then the distance is 0, its minimum value. If n_1 and n_2 are the same colour, but none of their N children can be matched to a same-coloured node (or alternatively, if one of n_1 or n_2 has N children, and the other has none), then the distance between n_1 and n_2 will be $N/(N+1)$. This property ensures that matching nodes of the same colour can never be worse than matching nodes of different colours, and that all incorrectly matched nodes will be on the outer fringe of the graph. This is desirable because we do not wish to continue matching nodes 'through a mistake'. To see this, consider a case where we can match a node in G_1 that has two children to a node in G_2 with two children. If the node in G_1 represents the 'plus' operation, and the node in G_2 is the 'distinct' operation (i.e. $\neq$), then it does not matter how well the child nodes can be matched. Context dictates that these two portions of graph serve different purposes.

Our method of averaging the edit distance is also desirable because it does not penalize nodes that have many children. If we dealt in total edit distance, then a pair of nodes with 8/10 successfully matched neighbours would have a higher distance than a pair of nodes with 1/2 matches.

The details of our distance algorithm are given by Figure 3. It can be summarized as a depth-first search that tries to find user-defined symbols, and stops when it reaches them. Two examples of this search are given by Figure 2. The Distance function should initially be called with two symbol nodes and a depth of 0. To prevent out-of-control graph expansion, the depths at which symbol, argument, and occurrence nodes can be expanded are tightly controlled. Line 6 ensures that symbol nodes can only be expanded at $depth = 0$ (i.e. initially). Argument nodes may only be fully expanded at $depth <= 1$ (line 8); otherwise, we only check which symbol is its parent. Occurrence nodes are replaced by their corresponding symbol node if $depth > 1$ (line 28), which effectively halts expansion. Combined, these depth limits have the effect of allowing full expansion for the initial symbol nodes and their child occurrence and argument nodes, but thereafter shutting down any expansion that could cross into separate GDL statements.

Briefly, we list some other notable features of the algorithm. Line 5 enforces a maximum search depth. From line 6, variable occurrence nodes are never expanded. Line 26 prevents cycles.

```
1: function DISTANCE(n1, n2, depth)
2:     if n1.colour = n2.colour then
3:         dist ← 0
4:         count ← 1
5:         if depth < MAX_DEPTH then
6:             if n1.colour is VAR_OCC OR
                   ↪ (n1.colour is SYMBOL AND depth > 0) then
7:                 pass
8:             else if n1.colour is ARG AND depth > 1 then
9:                 p1 ← SYMBOL parent of n1
10:                p2 ← SYMBOL parent of n2
11:                if p1.colour ≠ p2.colour then
12:                    dist ← dist + 1
13:                count ← 2
14:            else
15:                cDist, cCount ← ListDistance(n1.children,
                       ↪ n2.children, depth + 1)
16:                pDist, pCount ← ListDistance(n1.parents,
                       ↪ n2.parents, depth + 1)
17:                sDist, sCount ← ListDistance(n1.siblings,
                       ↪ n2.siblings, depth + 1)
18:                dist ← cDist + pDist + sDist
19:                count ← cCount + pCount + sCount + 1
20:        return dist/count
21:    else
22:        return 1
23:
24: function LISTDISTANCE(nList1, nList2, depth)
25:    for each node ∈ nList1 or nList2 do
26:        if node if it would form a cycle then
27:            remove node from its list
28:        else if node.colour is NON_VAR_OCC AND depth > 1 then
29:            replace node with its parent SYMBOL node
30:    assigned, tuples ← ∅
31:    totalDist, count ← 0
32:    for each n1 ∈ nList1 do
33:        for each n2 ∈ nList2 do
34:            dist ← Distance(n1, n2, depth)
35:            if dist = 0 then
36:                add n1 and n2 to assigned
37:                remove n1 from nList1 and n2 from nList2
38:                count ← count + 1
39:            else
40:                add (dist, n1, n2) to tuples
41:    sort tuples in increasing order by dist values
42:    while not all nodes ∈ assigned AND tuples is not empty do
43:        (dist, n1, n2) ← first element popped from tuples
44:        if n1 ∉ assigned & n2 ∉ assigned then
45:            add n1 and n2 to assigned
46:            remove n1 from nList1 and n2 from nList2
47:            totalDist ← totalDist + dist
48:            count ← count + 1
49:    for each remaining node ∈ nList1 or nList2 do
50:        totalDist ← totalDist + 1
51:        count ← count + 1
52:    return totalDist, count
```

Fig. 3. Algorithm for finding the distance of two rule graph nodes.

From lines 32 to 48, we are finding the distances of all pairs of nodes in two lists, and then greedily drawing pairs for nodes that have not yet been matched (like Kruskal's algorithm). Lines 35-38 represent an optimization that allows a pair to be assigned early, if it is a perfect match. Lines 49-51 assign a maximum distance of 1 to any nodes that were not matched.

C. Desirability Score

Using distance alone, it is sometimes possible for several pairs of symbol nodes to appear equally viable, which can be a problem when those pairs are incompatible with each other (i.e. want to map one node to different things). Since our mapping procedures are greedy, we provide additional information in the form of a desirability score (DS), to boost choices that are more likely to be correct. While searching graphs to find distance, we also track the total number of individual nodes successfully matched, and the number of those nodes that were previously mapped symbol nodes. Given two possible pairings with the same distance, we prefer the pair that successfully matched more nodes, because a larger graph expansion means that the pair is more likely to be mapped uniquely well. We also favour pairs with more previously-mapped neighbours. When a pair is added to the overall mapping, it is assigned a unique colour that is different from the generic colour for symbol nodes. Finding a match of these uniquely coloured nodes during a later graph search is rarer, and therefore more desirable, than a match among other colours. This feature is particularly useful for breaking ties among long chains of nodes that otherwise appear identical, like board coordinates.

The desirability score, DS, of matching nodes n_1 and n_2, is given by:

$$DS(n_1, n_2) = \alpha_D\left(\frac{Dist(n_1, n_2)}{D_{max}}\right) + \alpha_N\left(1 - \frac{Num(n_1, n_2)}{N_{max}}\right) + \alpha_A\left(1 - \frac{Assign(n_1, n_2)}{A_{max}}\right) \quad (2)$$

where $Dist$ is the distance function, D_{max} is the maximum distance among all pairs, Num is the total number of nodes successfully matched in the graph search for n_1 and n_2, N_{max} is the maximum number among all pairs, $Assign$ is the number of previously mapped (i.e. assigned) nodes matched during the distance search, A_{max} is its maximum among all pairs, and the α values are tunable weights. Generally, we want $\alpha_D > \alpha_N > \alpha_A$, but this isn't strictly necessary.

Finally, if we find more than one pair of nodes that score equally well and are incompatible with each other, we apply a flat penalty to the score of all such pairs. The intuition for this is that we would prefer to select a pair that is slightly more distant over one where we know that we are guessing. Later score updates may break the tie and lift the penalty.

D. Mapping Algorithms

After generating the rule graph for G_1, we have two methods for comparing and greedily selecting symbol pairs to add to the mapping, called MMap (Matrix Mapper) and LMap (Line Mapper).

Of the two, MMap is more thorough, but slower. After every assignment is made, it recalculates the desirability score for every pair of symbol nodes (which form a matrix), and selects the one with the lowest score to map next[3]. LMap employs the heuristic of Riesen et al.'s Greedy-GED algorithm [19], which requires calculating the distances for only two lines in the matrix that MMap generates. To do this, we must first select a node, n_1, from G_1. We make this selection using the Num and $Assign$ portion of the desirability score. The

[3]There is room for optimization here, since only the nodes neighbouring the most recently assigned pair actually need to be recalculated.

distance and DS are then calculated for n_1 and every symbol node in G_2. Among these, the node that produces the lowest DS is selected as n_2, and a distance and DS are found for every node in G_1 paired with n_2. From these pairs, the one with the lowest DS is added to the mapping. This second line of scores provides a second chance, in case the initial node selected from G_1 was a poor choice. However, LMap is still far more aggressive in its greed than MMap.

Whichever method is used, once a pair has been selected for mapping, the colour of both nodes is changed to a shared unique colour, and the process is repeated. Mapping stops when one or both graphs have run out of symbol nodes, or the best raw distance scores have exceeded a tunable threshold value. At this point, any remaining nodes are left unmatched, and assigned a distance of 1.

If we have multiple candidates for G_2, then this procedure can be repeated for each of them, and the one with the lowest overall distance should be considered the best mapping, and therefore the best option for transfer.

IV. Experimental Evaluation

The experiments reported in this section were designed to test the robustness of our MMap and LMap algorithms to unpredictable changes in the mapped game's rule graph. Each of them involve mapping some experimental game (G_1) onto a standard game that has previously been encountered and stored to disk (G_2). This recreates the experience of a transfer-bot, which would need to perform such a mapping before transfer learning. The majority of our experiments involve mapping onto either "8 Queens Puzzle - Legal Guided[4]"(which requires 8 chess queens to be placed on an 8×8 board without being able to attack each other) or standard Checkers, which can both be found in Stanford's base game repository [7]. These two games were chosen because they produce very differently sized rule graphs (given in Table I), and each has a number of useful variants that can also be found in Stanford's repository.

For each trial, we randomized the order of the nodes in G_1's rule graph. This does not change the structure of the graph, but ensures that the order in which GDL statements are presented in the rule set cannot factor into the mapping process. A mapping algorithm that relied on this ordering could easily be defeated, since the order of statements can be changed without affecting the underlying game.

Numeric symbols have been omitted from correctness percentages, where a numeric symbol is one that represents a number, or part of a number-like sequences (like 'm1', 'm2', 'm3', etc.). We have made this choice because numbers are defined in a structured, repetitive way that allows them to be discovered by simpler methods [14], [15], and they often significantly outnumber all other nodes. Checkers, for instance, defines 202 numeric symbols, compared to only 50 other non-variable symbols, which we are more interested in mapping.

[4]"Legal Guided" refers to extra rules that prevent a bot from making illegal moves. In "Unguided" versions of the game, illegal moves are allowed, but will result in a score of 0 upon termination.

Cross-domain mapping is a relatively underdeveloped area of research for GGP, so we do not have an established baseline to compare against. We will, however, include results for a "Myopic" mapping algorithm that functions identically to LMap, but has a maximum search depth of 1. This limits its view to the immediate neighbourhood around each symbol node, and makes it essentially equivalent to the original Greedy-GED algorithm [19].

Mean values reported have been averaged over 20 trials run with different random seeds. We used a maximum search depth of 5, a penalty for incompatible pairs of 0.1, a distance threshold of 0.5, and parameter values $\alpha_D = 0.8$, $\alpha_N = 0.18$, and $\alpha_A = 0.02$. All code was written in Java, and all experiments were run as single-threaded processes on a Windows 10 machine with a 2.8 GHz i7 processor.

In this section, we detail the methodology for three different kinds of quantitative evaluation. Section V discusses results, and additionally notes several informal experiments that cannot be readily evaluated.

1) Self-Mapping: As a baseline, we begin by mapping the rule graphs of various games onto themselves, unaltered. (I.e. If G_1 is Checkers, then G_2 is also Checkers.) Although we know that a perfect matching must exist, this process of self-mapping is not guaranteed to succeed flawlessly, as some symbol nodes may appear to be identical when considering only a finite neighbourhood around them. Since we know which symbols ought to be matched together, the correctness of a mapping is simply given by the percentage of these matches that were actually made.

2) Adding or Removing Nodes: We continue self-mapping, but introduce unpredictable changes to G_1 by either deleting or duplicating nodes chosen randomly. Deletion has the effect of reducing the information present in G_1, while duplication adds noise that can be misleading. To delete a node, we remove all of its in- and out-edges, making it completely inaccessible from elsewhere in the graph. To duplicate a node, we create a second instance of it and duplicate all in- and out-edges. In both cases, we target occurrence nodes[5] because they provide essential information to our mapping algorithms, and because a change to one occurrence node is directly comparable to one change in the GDL code. In particular, removing an occurrence node is akin to crossing out a single symbol in GDL. (Duplicating a node does not have a similarly intuitive analogy, but is essentially the inverse operation.)

In general, this may leave functions without arguments or arguments without parents, so the resulting graph no longer represents a syntactically correct game. We are therefore only approximating the actual GDL changes that a GGP bot might encounter. However, this approach is useful for its ability to alter a rule graph by any amount in a way that cannot be predicted by a bot's creator in advance. Since we are ultimately still self-mapping, correctness can be evaluated in the same way as previously.

[5]We also tested duplicating/deleting any nodes that were not symbol nodes. Results followed the same general patterns observed when altering only occurrence nodes.

TABLE I
SELF-MAPPING STATISTICS FOR GAMES OF VARYING COMPLEXITY.

Game	# Nodes	Time to Build Graph (ms)	Myopic Self-Mapping		MMap Self-Mapping		LMap Self-Mapping	
			% Correct	Time (ms)	% Correct	Time (ms)	% Correct	Time (ms)
8 Queens, Gd.	464	1.81 ±1.64	80.63 ±9.25	5 ±7	**100.00** ±0.00	224 ±140	**100.00** ±0.00	59 ±28
Tic-tac-toe	469	1.75 ±1.83	85.33 ±10.24	3 ±3	**88.00** ±10.24	133 ±70	85.33 ±10.24	46 ±24
Connect Four	553	1.86 ±1.90	96.79 ±3.55	2 ±3	**100.00** ±0.00	125 ±60	**100.00** ±0.00	54 ±27
Rubik's Cube	1521	2.85 ±5.64	50.31 ±5.29	11 ±9	**100.00** ±0.00	8612 ±451	**100.00** ±0.00	535 ±238
Checkers	3990	4.73 ±7.42	67.10 ±5.92	49 ±27	**100.00** ±0.00	43680 ±4166	**100.00** ±0.00	1371 ±316
Chess	5668	7.88 ±13.30	65.71 ±4.20	57 ±26	**97.14** ±1.21	86964 ±4400	95.18 ±2.34	3612 ±323

3) Mapping to Game Variants: To ensure that mapping can be performed between functional games that are different in meaningful ways, we examine the families of games that include 8 Queens and Checkers. Within a family, symbol names and much of the game logic are shared, but games are different by some combination of altered board size, board topology, or rule set. For example, a game within the Checkers family is Checkers, played on a Torus, where pieces must jump if they are able. We assign either 8 Queens (Guided) or Checkers to G_2, and their respective variants to G_1.

Although our mapping algorithms cannot make use of the shared symbol names due to obfuscation, these names are useful for evaluation. We calculate the correctness of a mapping by first finding the set of symbols whose names appear in both G_1 and G_2, and then finding the fraction of those symbols that are matched correctly. This approach would not work if our games were not explicitly part of a family, as two symbols could share a name without actually representing the same concept. The problem of evaluating mappings between more distantly related games is discussed further in Section V-B.

V. RESULTS AND DISCUSSION

A. Main Results

1) Self-Mapping: Table I gives results for the self-mapping of various games. In addition to 8 Queens and Checkers, a few other well-known games were chosen at varying levels of complexity. Although the time taken to build a rule graph does scale with the complexity of that graph, this time was less than 10 milliseconds for all games tested. Since it is insignificant compared to the time taken for mapping, we will ignore graph generation time, moving forward.

The Myopic mapper is (naturally) very fast, and performs fairly well for simple games, but correctness drops sharply for games that are more complex. This is sensible, as larger rule graphs provide more opportunity for nodes to be similar in their immediate neighbourhoods. MMap and LMap compare well to each other with regard to mapping accuracy. They score less than 100% in the same places, and for the same reasons. In Tic-tac-toe, they sometimes map 'X' to 'O' and/or 'row' to 'column'. This is understandable, as these symbols serve very similar purposes and appear identical within the bounds of our search, even though they would be differentiable if that search were extended to the entire graph. We observe a similar phenomenon for Chess, where symbols representing the two directions for diagonal checking, as well as rook/queen or bishop/queen attack symbols may be confused.

Run time clearly favours LMap over MMap. While LMap runs in less than 4 seconds for all games tested, MMap takes ∼44 seconds for Checkers and ∼87 seconds for Chess. This is problematic for GGP, where a bot can expect to have no more than 30 seconds to 1 minute for initialization. This limits the application of MMap to relatively simple games only, barring further optimization.

2) Adding or Removing Nodes: From Figure 4, it is immediately clear that both MMap and LMap (with a maximum search depth of 5) outclass a Myopic mapper (with a maximum depth of 1) in terms of mapping accuracy. This was true when 0 nodes were removed/duplicated, and remained true for all values that we tested.

In the node removal experiment, we varied the number of deletions from 0 up to the total number of occurrence nodes in the rule graph. Neither MMap nor LMap dropped to 0% accuracy, even when all occurence nodes were removed because they were able to glean some information using argument nodes, exclusively. The Myopic mapper was rendered completely ineffective at this stage. For both games, MMap and LMap retained an accuracy of more than 70% until over 50% of occurrence nodes had been removed.

Our node duplication experiment similarly shows that both MMap and LMap produce far better accuracy that a Myopic mapper for any number of occurrence nodes duplicated. Here, we also see a gap between MMap and LMap that is present in both games, but most pronounced for Checkers. It indicates that MMap is more robust than LMap to the addition of random misleading information, though this comes at the cost of a much higher execution time.

3) Mapping to Game Variants: Table II gives results for mapping 8 Queen variants onto '8 Queens, Legal Guided' (hereafter, just '8 Queens'), and Checkers variants onto Checkers. The percentage of non-numeric symbols shared serves as a rough, but incomplete measure of similarity between games. Although a symbol may be shared, its usage can be different. We see this in the results for '31 Queens, Guided'. Despite all non-numeric symbols being shared with 8 Queens, it was the only 8 Queens variant that caused both MMap and LMap to score less than 100% in mapping accuracy. This occurred because 31 Queens features many more numeric symbols. Though they are not directly included in accuracy calculations, the existence of these symbols caused mis-mappings among other symbols that consume them as arguments.

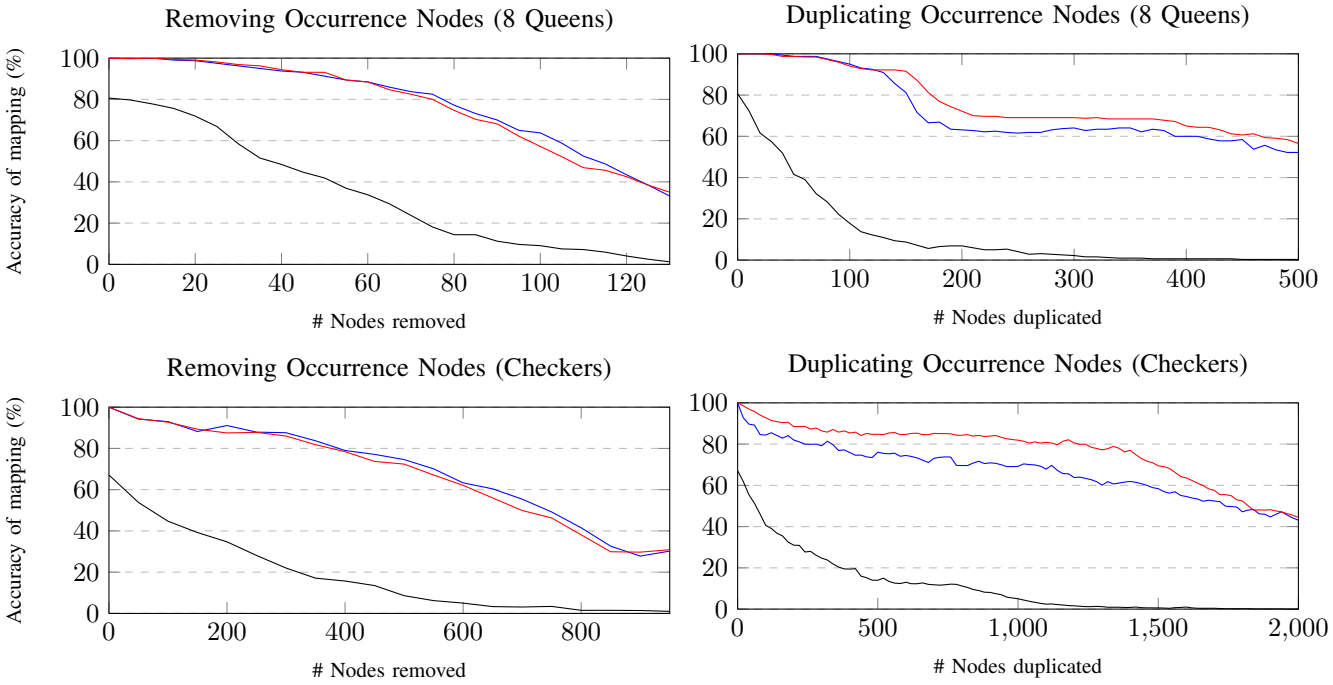

Fig. 4. Mapping accuracy for alterations of 8 Queens (top) and Checkers (bottom). MMap in red, LMap in blue, Myopic in black.

TABLE II
MAPPING STATISTICS FOR THE 8 QUEENS AND CHECKERS FAMILIES.

Game Variant	% Sym. Shared	Myopic		MMap		LMap	
		% Correct	Time (ms)	% Correct	Time (ms)	% Correct	Time (ms)
6 Queens, Unguided	93.33	89.29 ±8.60	5 ±8	**100.00** ±0.00	166 ±95	**100.00** ±0.00	54 ±24
8 Queens, Unguided	93.33	89.29 ±8.60	3 ±4	**100.00** ±0.00	203 ±110	**100.00** ±0.00	41 ±17
12 Queens, Unguided	93.33	75.71 ±6.55	4 ±5	**100.00** ±0.00	256 ±142	**100.00** ±0.00	62 ±34
16 Queens, Unguided	93.33	75.71 ±6.55	4 ±5	**100.00** ±0.00	300 ±96	**100.00** ±0.00	56 ±30
31 Queens, Guided	100.00	67.50 ±8.05	6 ±8	**87.50** ±0.00	584 ±197	68.50 ±0.00	123 ±80
Checkers, Small (6X8)	97.96	61.15 ±4.38	45 ±28	**97.92** ±0.00	36586 ±674	**97.92** ±0.00	1279 ±231
Checkers, Tiny (4X8)	95.83	69.67 ±5.43	36 ±14	**97.83** ±0.00	35996 ±640	**97.83** ±0.00	1214 ±210
Checkers, Must-Jump	84.62	41.25 ±4.15	22 ±14	**100.00** ±0.00	36446 ±280	86.02 ±4.84	2251 ±211
Checkers, Cylinder, Must-Jump	84.62	40.00 ±3.25	24 ±14	**100.00** ±0.00	34373 ±525	81.82 ±7.91	2351 ±172
Checkers, Torus, Must-Jump	84.62	40.00 ±3.25	24 ±14	**100.00** ±0.00	34618 ±557	80.80 ±6.95	2334 ±221

In general, we see the same patterns from this experiment as have been previously established. The Myopic mapper crumbles quickly as complexity is increased, and as symbol matches become less clear-cut. LMap is considerably faster than MMap, but also more prone to errors. This is particularly evident when looking at the Checkers variants, where MMap scores 100% accuracy across the worst cases for LMap. On the other hand, MMap takes more than 30 seconds for every Checkers variant, which is likely too slow for use in GGP. It remains to be seen if LMap's mapping accuracy is low enough to impede good transfer in a GGP bot.

B. Additional Results, Limitations, and Future Work

It is clear that an important next step for this line of research is to test its compatibility with various methods of transfer learning, like those discussed in Section II-C. This would allow us to answer questions about the effectiveness of transfer at various levels of mapping accuracy. Additionally, we could analyze mappings between games that are not as closely related, where concepts may be shared even when symbol names are not. For example, we know that Checkers and Chess are both games played on a board, where players alternate turns, moving one piece at a time. If we try mapping Chess to Checkers (with LMap, in this case), some of these intuitive connections are borne out (e.g. board properties, like rank and file, or capturing in Chess being mapped to jumping in Checkers). On the other hand, some are dubiously useful (e.g. knight movement mapped to king movement), or obviously wrong (e.g. the en passant capturing rule mapped to the 'greater than' operation). For this to be a useful mapping overall, it must not only produce positive transfer, but also provide a larger benefit than spending the same amount of time on self-play. The most direct way to test this is to build a transfer-bot and have it play against non-transfer counterparts.

Before conducting transfer, a bot must first decide which game to transfer onto, given the choice of every game that it has previously encountered. Here, mapping (LMap, in particular), serves another important function. Because we are keeping track of the edit distance for every pair in the mapping, it is straightforward to produce an overall distance for two

games that serves as a measure of similarity. Some examples of similarity between standard Checkers and various other games are given by Table III. Intuitively, it seems sensible that Checkers is closest to its own variants, farther from Chess, and farther still from games like Connect Four, although we cannot give an objective evaluation of these distance values.

If a bot is able to run a mapping between its current task and every game in its knowledge base, then it can choose to transfer from the one with the lowest average distance. For this kind of usage, even an execution time of 4 seconds becomes problematic, but there are ways that LMap can be optimized. Since LMap is greedy, and the distance for a mapped pair is fixed once that pair has been selected, we can terminate LMap early when it becomes mathematically impossible for the current mapping to produce a lower average distance than the previously seen best. We could also consider applying more aggressive heuristics, like shortening the maximum search depth. Since we only care about approximating the overall distance, some mapping quality could be sacrificed with the intention of doing a more complete mapping for the game that is ultimately selected for transfer. This approach could also be necessary for scaling to games larger than Chess.

As the library of known games for transfer grows, it may be useful to cluster them hierarchically. For example, all of the Checkers variants might belong to one cluster, and that cluster might be grouped with others at the next level. If a given game matches poorly with a representative from a cluster, time could be saved by declining to check the cluster's other members. This system also provides some relief to the problem of games that are the same, but are described differently enough to inhibit a good mapping. We will not be able to recognize the similarity of the descriptions, but can store all of them within the hierarchy without significantly hampering execution time.

Another avenue for future research is the application of MMap and LMap to GGP frameworks other than GDL, such as Ludii [20]. Since it is designed for clear, concise game descriptions, we are optimistic that mappings could be both more effective and faster to produce.

Beyond game playing in the abstract, a practical application of this work will be to the transfer learning problems present in smart home technologies for persons suffering from dementia. In such problems, complex cognitive models of people may be built that help a system to provide timely and appropriate assistance, but these models will be structured around one particular domain (e.g. kitchen activities). We are particularly interested in how these models may be mapped to other activities, easing the adoption of the technology more widely across the home than is currently possible.

VI. Conclusion

We have developed a method for approximating an edit distance between nodes in two different rule graphs, and have applied it in two methods for mapping symbols from a source game to a target game of differing domain. Our evaluation shows that, while both methods achieve a high mapping accuracy for games that have been altered in unpredictable ways,

TABLE III
Mean distances from standard checkers.

Game	MMap Distance	LMap Distance
Checkers	0.0000 ±0.0000	0.0000 ±0.0000
Checkers, Must-Jump	0.0929 ±0.0000	0.1135 ±0.0011
Checkers, Torus, M-J	0.0940 ±0.0000	0.1134 ±0.0007
Chess	0.4270 ±0.0043	0.4172 ±0.0034
8 Queens, Guided	0.7520 ±0.0000	0.7394 ±0.0001
Connect Four	0.7567 ±0.0037	0.7482 ±0.0003

MMap is somewhat more robust, but LMap is significantly faster. In the GGP setting, where time is at a premium, LMap is generally viable as a starting point for transfer. With further optimization, MMap may become generally viable as well, but for now, remains useful for low-complexity games.

Acknowledgment

This work was generously supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] D. Silver *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[2] A. Goldwaser and M. Thielscher, "Deep reinforcement learning for general game playing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 34, no. 02, 2020, pp. 1701–1708.

[3] M. Świechowski, H. Park, J. Mańdziuk, and K.-J. Kim, "Recent advances in general game playing," *The Scientific World Journal*, 2015.

[4] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the aaai competition," *AI magazine*, vol. 26, no. 2, pp. 62–62, 2005.

[5] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth, "General game playing: Game description language specification," 2008.

[6] M. Genesereth and M. Thielscher, "General game playing," *Synthesis Lectures on A.I. and Machine Learning*, vol. 8, no. 2, pp. 1–229, 2014.

[7] A. L. Sam Schreiber, "The general game playing base package," 2018. [Online]. Available: https://github.com/chuchro3/ggp-base

[8] G. J. Kuhlmann, "Automated domain analysis and transfer learning in general game playing," Ph.D. dissertation, Uni. of Texas at Austin, 2010.

[9] G. Kuhlmann and P. Stone, "Graph-based domain mapping for transfer learning in general games," in *European Conference on Machine Learning*. Springer, 2007, pp. 188–200.

[10] G. Jiang, L. Perrussel, D. Zhang, H. Zhang, and Y. Zhang, "Game equivalence and bisimulation for game description language," in *Pacific rim intl. conf. on artificial intelligence*. Springer, 2019, pp. 583–596.

[11] B. Falkenhainer, K. D. Forbus, and D. Gentner, "The structure-mapping engine: Algorithm and examples," *Artificial intelligence*, vol. 41, no. 1, pp. 1–63, 1989.

[12] M. Klenk and K. Forbus, "Cross domain analogies for learning domain theories," Northwestern Univ, Evanston, IL, Qualitative Reasoning Group, Tech. Rep., 2007.

[13] T. Hinrichs and K. D. Forbus, "Transfer learning through analogy in games," *Ai Magazine*, vol. 32, no. 1, pp. 70–70, 2011.

[14] G. Kuhlmann and P. Stone, "Automatic heuristic construction for general game playing." in *AAAI*, 2006, pp. 1883–1884.

[15] S. Schiffel and M. Thielscher, "Fluxplayer: A successful general game player," in *Aaai*, vol. 7, 2007, pp. 1191–1196.

[16] D. M. Kaiser, "Automatic feature extraction for autonomous general game playing agents," in *Proc. of the 6th international joint conference on Autonomous agents and multiagent systems*, 2007, pp. 1–7.

[17] M. Kirci, N. Sturtevant, and J. Schaeffer, "A ggp feature learning algorithm," *KI-Künstliche Intelligenz*, vol. 25, no. 1, pp. 35–42, 2011.

[18] A. Hufschmitt, J.-N. Vittaut, and J. Méhat, "A general approach of game description decomposition for general game playing," in *Computer Games*. Springer, 2016, pp. 165–177.

[19] K. Riesen, M. Ferrer, and H. Bunke, "Approximate graph edit distance in quadratic time," *IEEE/ACM transactions on computational biology and bioinformatics*, 2015.

[20] E. Piette *et al.*, "Ludii–the ludemic general game system," *arXiv preprint arXiv:1905.05013*, 2019.