Graph-Based Mapping for Knowledge Transfer in General Game Playing

by

Joshua Jung

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2024

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Martin Müller Professor, Dept. of Computing Science, University of Alberta
Supervisor:	Jesse Hoey Professor, Cheriton School of Computer Science, University of Waterloo
Internal Members:	Kate Larson Professor, Cheriton School of Computer Science, University of Waterloo
	Peter van Beek Professor Emeritus, Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Arie Gurfinkel Professor, Dept. of Electrical and Computer Engineering, University of Waterloo

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

General game playing (GGP) is a field of reinforcement learning (RL) in which the rules of a game (i.e. the state and dynamics of an RL domain) are not specified until runtime. A GGP agent must therefore be able to play any possible game at an acceptable level given an initialization time on the order of seconds. This time restriction promotes generality, precludes the use of the deep learning methods that are popular in the RL literature, and has led to the widespread use of Monte Carlo Tree Search (MCTS) as a planning strategy. A typical MCTS planner builds a search tree from scratch for every new game, but this leaves usable information on the table. Over its full history of play, an agent may have previously encountered a similar game from which it could draw insights into its current challenge. However, recognizing similarity between games and effectively transferring knowledge from past experience is a non-trivial task.

In this thesis, we develop methods for automatically identifying similar features in two related games by finding an approximated edit distance between the graphs generated from their rules. We use that information to guide MCTS in one game with general heuristics initialized via transfer from a previously played game. Despite the computational cost of doing so, we show that the more efficient search granted by this approach can lead to better performance than either UCT (a standard method of MCTS) or a non-transfer MCTS agent with access to the same general heuristics. We examine the circumstances under which transfer is most effective, and also identify and create solutions for the cases where it is not.

Acknowledgements

I am extremely grateful to my supervisor, Prof. Jesse Hoey, whose support and patience have never wavered. He took a risk in allowing me to pursue the topic that I was most passionate about, and I will always be thankful for his trust in me. I would also like to thank my examination committee for their advice, feedback, and time.

I am thankful for my family - my parents, sister, grandmothers, cousins, aunts, and uncles, all. Every visit home, I have been met with an outpouring of support, and I have never, even for a moment, doubted that they believed in me.

Finally, I would like to thank my dear friends and labmates for their support and welcome distraction, with special thanks to Ryan Hancock, Deepak Rishi, Alexander Sachs, Nolan Shaw, and Ian Swintak.

Table of Contents

List of Tables ix

List of Figures x

1 Introduction 1

1.1 Contributions . . . 2
1.2 Motivation . . . 3
1.3 Thesis Summary . . . 5

2 Background and Related Work 8

2.1 Reinforcement Learning . . . 8
2.1.1 Temporal Difference Learning . . . 10
2.1.2 MCTS . . . 10
2.1.3 State Abstraction . . . 12
2.1.4 Reward Shaping . . . 13
2.1.5 Lifelong Learning . . . 14
2.1.6 Deep Reinforcement Learning . . . 15
2.2 General Game Playing . . . 15
2.2.1 General Game Playing Competitions . . . 16
2.2.2 Game Description Language . . . 18
2.2.3 Rule Graphs . . . 20

2.2.4 General Heuristics . . . 21
2.3 Transfer Learning . . . 23
2.3.1 Policy Transfer . . . 24
2.3.2 Value Function/Heuristic Transfer . . . 25

3 Mapping **27**
3.1 Background . . . 28
3.1.1 Rule Graphs . . . 28
3.1.2 Related Work in Mapping and Transfer . . . 30
3.2 Method . . . 30
3.2.1 Rule Graph Generation . . . 30
3.2.2 Approximate Edit Distance . . . 31
3.2.3 Desirability Score . . . 35
3.2.4 Mapping Algorithms . . . 36
3.3 Experimental Evaluation . . . 37
3.4 Results and Discussion . . . 39
3.4.1 Main Results . . . 39
3.5 Conclusion . . . 45

4 Transfer via General Heuristics **46**
4.1 Background . . . 47
4.2 Method . . . 48
4.2.1 General Heuristic Calculation . . . 48
4.2.2 Guided Monte Carlo Tree Search . . . 51
4.2.3 Archived Information . . . 51
4.2.4 Rule Graphs and Symbol Mapping . . . 53
4.2.5 Data Storage . . . 54
4.2.6 A Note on Direct State Comparison . . . 60

4.3 Experimental Evaluation . . . 61
4.3.1 Checkers Variants . . . 61
4.3.2 Breakthrough Variants . . . 62
4.3.3 Results . . . 64
4.4 Discussion . . . 65
4.5 Conclusion . . . 68

5 Greater Generality **69**
5.1 Source Game Selection . . . 70
5.2 Board Heuristics . . . 72
5.2.1 Method . . . 73
5.2.2 Evaluation . . . 76
5.2.3 Results . . . 78
5.3 Negative Transfer Protection . . . 83
5.3.1 Method . . . 83
5.3.2 Evaluation . . . 85
5.3.3 Results . . . 86
5.4 Single-Player Games . . . 90
5.4.1 Properties of Single-Player Games . . . 91
5.4.2 Vacuum Checkers . . . 92
5.4.3 Evaluation . . . 94
5.4.4 Results . . . 96

6 Conclusion **98**
6.1 Future Work . . . 99
6.2 Closing Thoughts . . . 101

References **102**

List of Tables

2.1 Rule graph sizes for games of varying complexity. 20

3.1 Self-mapping results 40
3.2 Mapping results for game variants 43
3.3 Mean distances from standard checkers. 44

4.1 Checkers results 63
4.2 Breakthrough results 64
4.3 Standard deviation in selected heuristic r-values 65

5.1 Shallow LMap distances 71
5.2 Connect Four results with board heuristics 79
5.3 Checkers results with board heuristics 81
5.4 Breakthrough results with board heuristics 82
5.5 Connect Four partial swap results 86
5.6 Connect Four full swap results 87
5.7 Checkers full swap results 88
5.8 Reversed Connect Four full swap results 89
5.9 Reversed Checkers full swap results 90
5.10 Vacuum Checkers results 96

List of Figures

2.1 Obfuscation of a game description 17
2.2 A numeric chain in GDL 18
2.3 GDL to rule graph example 19

3.1 Distance algorithm (Part 1) 32
3.2 Distance algorithm (Part 2) 33
3.3 Distance algorithm visualization 34
3.4 Mapping results for adding/removing nodes 42

4.1 Rollouts over time 52
4.2 Rule graph file 56
4.3 Raw heuristic data file 57
4.4 Regression value file 58
4.5 SI-GMCTS mobility r-value distribution 66

5.1 Edge Connect Four 77
5.2 Vacuum Checkers layouts 95

Chapter 1

Introduction

Games have long been an important test bed and source of grand challenges for Artificial Intelligence (AI) research. For machines, their strict rules and well-defined state transitions provide predictable conditions for evaluating new methods. For humans, games are challenging, thought-provoking, and of course, fun. Today, AIs like Deep Blue, Watson, and AlphaGo remain household names, owing to the spectacle of their mastery over games that everyday people play, enjoy, and know to be difficult.

Progress in this field may be marked by the complexity of the games being examined and the level of dominance achieved by the artificial agents (sometimes called AIs or bots) that play them. Simpler games, like Connect Four [5] and Checkers [65], have been solved such that no other player can possibly win. More complex games, such as Chess [15] and Go [70] are not fully solved, but can be played by an AI to a level that far surpasses the best human players. Each of these milestones represent a triumph of engineering that captures the imagination and demonstrates how far AI research has come, but in it, they also expose a fault. These champion agents are highly specialized. AlphaGo is a master of Go, but can not play Go Fish. Its successor, AlphaZero, can learn to play a variety of games at a world-class level, but requires hours to weeks of training on powerful banks of machines for each one. This kind of resource expenditure is simply not appropriate for many of the 'games' that make up the world around us. Here, a 'game' does not necessarily refer to a recreational activity, but rather, a problem in which one must navigate from some initial state to a terminal state via a set of well-defined rules. This definition encompasses a broad variety of tasks, and an agent that could be presented with any such task and very quickly perform at an acceptable level would be an extremely useful multi-purpose assistant. It would come closer to achieving artificial general intelligence than any bot trained exclusively for a single game, like Chess.

General Game Playing (GGP) is a field of study whose researchers endeavour toward this goal. In GGP, an agent must be able to play any game presented to it, without ever having seen it before. It has a short window for initialization (up to a few minutes) and thereafter must submit moves every few seconds until the game terminates[1]. These strict time restrictions are intended to force a more general approach to game playing (hence the name). There simply isn't time to train large deep neural networks for every problem, and as a result, state-searching algorithms still flourish in this domain.

Monte Carlo Tree Search (MCTS), in particular, has proven to be very effective in GGP. Its greatest asset is its ability to see terminal states very quickly, which allow it to detect promising strategies even in domains with very sparse reward structures. As a result, MCTS agents perform relatively well learning every game from scratch, without carrying knowledge gained in one game to the next. However, this leaves potentially useful information on the table that could be used to push them even farther. Experience as a human player tells us that analogy is a powerful tool for learning. If one already understands the game of Checkers, it should be relatively simple to transition to Checkers on a larger board. In a realm where time is so limited, a better understanding of the game means more efficient search and better performance. When knowledge gained in one game is applied to another in this way, we say that the knowledge has been *transferred*, and refer to the AI as a *transfer agent*. Over a lifetime of service, a GGP agent should accumulate a wealth of knowledge and leverage it to become more adept at quickly adapting to new challenges via transfer. This is important because it is what humans do, and because it is wasteful not to do so.

In this thesis, we develop methods for automatically identifying similar features between two related games, and using that information to guide MCTS via general heuristics. Despite the computational cost of doing so, we show that the more efficient search granted by this approach can lead to better performance than either UCT (a standard method of MCTS) or a non-transfer agent with access to the same general heuristics. We examine the circumstances under which transfer is most effective, and also identify and create solutions for the cases where it is not.

1.1 Contributions

The contributions made by this thesis may be summarized as follows:

[1]The exact periods of time given for initialization and turn-taking are variable, and may be chosen by the organizer of a GGP event. 60 seconds for initialization, and 15 seconds for every subsequent turn are reasonable default values, taken from Stanford's ongoing online tournament.

- A method for approximating an edit distance between nodes in the graphs generated from the rules of two different games.
- Two algorithms, LMap and MMap, that use this edit distance to find a mapping between the most similar symbols in those games.
- A system for organizing and storing data needed for transfer.
- A modular system for calculating general heuristics, and a modified version of UCB1 that is influenced by those heuristics, weighted by the strength of their correlation with reward.
- TI-GMCTS, a GGP agent that uses transferred knowledge to guide MCTS, using all of the contributions above.
- Ancillary systems for TI-GMCTS, including source game selection and negative transfer protection.

The first two of these contributions were published at the 2021 Conference on Games [36] with co-author Jesse Hoey, who provided supervisory assistance and proofreading.

In sum, these contributions allow us to achieve autonomous transfer within the time scales permitted by GGP. On its own, we believe this to be a novel result within the research area. In incorporating these methods with MCTS, we are certainly the first, which is important for the reasons discussed in Section 1.2.

This thesis makes use of the original Game Description Language [52] (described in detail in Section 2.2.2), which limits our focus to deterministic games (although the methods we describe are agnostic to non-determinism). Additionally, we consider only single-player games and two-player games in which players alternate turns (which need not be zero-sum). Application to other languages is considered in Section 6.1.

A repository of the code associated with this thesis can be found at `https://github.com/jdajung/ggp-transfer`.

1.2 Motivation

GGP: We begin by addressing a question that is unavoidable given the state of modern AI: GGP is a field of research that was created before the rise in popularity of deep learning. Does it still have a place in a world with AlphaZero and large language models? We believe

the answer to be yes, unambiguously. It is difficult to apply deep reinforcement learning methods to GGP because of the extremely limited time and computational power that such an agent would have to train. For this reason, GGP lends itself well to problems that are on a smaller scale than grand challenges in AI, problems for which it would be inappropriate to undertake the cost of training a model like AlphaZero, or even the fine-tuning of such an agent.

We refer to 'problems' and 'games' interchangeably in this context, because games in GGP need not be fun. Due in part to its history of tournament-style competitions, the GGP literature heavily features two-player board games, but we are not limited to them. A GGP agent is, at its heart, a general-purpose problem-solver.

For example, consider a personal assistant application. ChatGPT has shown that there is a market for an agent capable of answering any question, including performing small problem-solving tasks. However, this is a task for which large language models are not well suited, since their responses are not beholden to the rules of the problem, and they have a propensity to offer nonsense solutions or lies. It is a niche that requires a quick response at low computational cost, within a well-defined set of rules, which is exactly the domain of GGP.

Games for Fun: Having just described GGP as a good fit for serious tasks, we now make the case that there is real value in playing games for fun, as well, and discuss the reasons that we are (mostly) concerned with two-player board games in this work. Part of this value is in communication. For the same reasons that board games attracted public notoriety as grand challenges, they are useful in demonstrating ideas, even to people with no prior knowledge of GGP. These games are familiar and easy to visualize. We can quickly develop our own heuristics for playing them that give insight into which general heuristics are likely to be most effective. We will discuss several ways in which board games can by transformed to produce variant games, and these transformations, as well as their implications on state space and strategy, are easiest to intuit and understand when they are applied to a well-known base game. Games also make progress easy to measure, since new techniques can be tested against other bots, as well as human players.

There is also economic value in purely recreational games. Video games frequently feature AIs at various levels of difficulty for human opponents to play. Having stock GGP agents available for this purpose would save the cost of having a developer code these bots from scratch. Although a GGP agent would not be likely to play the game as well as, for example, a well trained AlphaZero bot, it is not normally desirable for video game AIs to be unbeatable. Rather, they are meant to impose a variable level of challenge to the

player, which, for a GGP agent, can reasonably be tuned by adjusting the amount of search permitted.

Finally, there is value in pursuing fun for its own sake. Passion for a topic is an excellent motivator for pushing it forward, and, to set aside formality for a moment, it has been my privilege to work on one for which I am genuinely excited. I hope that, in reading this research, you are also able to experience some of that enjoyment.

MCTS: If deep learning is currently limited in its applicability to GGP, then GGP remains a relevant research area for other methods to flourish. As previously alluded, MCTS was a dominant method in the annual GGP competitions [79], and is deeply rooted in the GGP literature. There is therefore good reason to build our new methods on it (as opposed to another search or reinforcement learning algorithm), since doing so allows our work to be applied more easily to existing projects, and integrated with methods that we have not used. It also allows us to use UCT (described in Section 2.1.2) to be used as a strong, well-understood baseline for comparison that shares all of the same basic code as our experimental agents. This prevents differences in coding efficiency from impacting our findings.

Transfer: We discuss some related works applying transfer to GGP or related problem areas in Section 2.3, but overall, it remains an underdeveloped area of research, and we are not aware of any GGP agents using transfer in competition, or an environment that mimics one. Instead, transfer has typically either been unconstrained by time, or not fully autonomous (e.g. by requiring a hand-made mapping of similar symbols, or a hand-made set of rule graph templates for comparison).

Additionally, none of these methods show a successful applications of transfer to MCTS. Kuhlmann's work [46] includes an attempt to transfer knowledge to or from UCT, but was not successful in doing so. This makes our work uniquely useful as the first application of transfer to GGP's most dominant method. An agent's previous knowledge is an untapped resource, and we are providing a way to exploit it.

1.3 Thesis Summary

Chapter 2 introduces necessary background and discusses related work, with a focus on the GGP literature.

In Chapter 3, we present novel algorithms, LMap and MMap, for autonomously finding a mapping for the symbols of the current game (the *target* game) to those that are most similar in a previously played game (the *source* game), where 'symbols' include features like state variables and possible actions. This is necessary not only because there is a chance that similar features might not share names across two different games, but because GGP enforces obfuscation of the game rules to ensure that they do not. (Obfuscation is described in greater detail in Section 2.2.2.) Obtaining a mapping is a necessary prerequisite for transfer because many (all but one) of the general heuristics that we apply require knowledge of like symbols. For example, a reasonable heuristic in Checkers might be to reduce the number of pieces of your opponent's colour, but to make use of this information, you have to know which symbols correspond to those pieces.

After describing LMap and MMap, 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.

Using LMap to obtain a symbol mapping, Chapter 4 presents a technique for automatically transferring general heuristic knowledge between distinct, but related, games. We describe in detail how to calculate each of the general heuristics used, and provide a method for combining them into one evaluation function, weighted by their relative strength. We show how these heuristics are initialized, either through transfer or simulation, and we compare the performance of the corresponding agents (TI-GMCTS and SI-GMCTS) in six variants of Checkers and variants of Breakthrough with varying state/action spaces. We find that transfer can improve the quality of game-independent heuristics to produce better performance in games within the GGP framework, especially when initialization time is short.

Chapter 5 pushes TI-GMCTS toward greater generality by tackling previously deferred or entirely new problem areas. First, we address the problem of source game selection. By using LMap at a low search depth, we show that it is possible to determine a most similar source game to the current target game on a time scale that is compatible with GGP. We then demonstrate modularity in our handling of general heuristics by dropping in a new family of board heuristics, and using them to outperform SI-GMCTS and UCT in Connect Four, a game that our original heuristics were not well suited for. Next, we present a method for mitigating the effects of negative transfer, and use it to obtain better performance on inverted goal game variants, previously a worst-case scenario for TI-GMCTS. We conclude the chapter by addressing single-player games, and show that it is possible for transfer to be effective from a two-player game to a single-player game when the factors that make a position good are well aligned.

Finally, Chapter 6 concludes, and discusses possible directions for future work.

Chapter 2

Background and Related Work

In this chapter, we broadly discuss relevant methods and reference related works. When more specific detail of a method is required, it is provided in the chapter where it is needed.

2.1 Reinforcement Learning

GGP can viewed as a restricted form of reinforcement learning (RL). In this section, we introduce useful vocabulary and concepts from standard RL. We will also discuss some of the idiosyncrasies of GGP as they are relevant, and the rest in Section 2.2.

Under the RL paradigm, an agent is placed into an environment and given a reward signal for every interaction with that environment. This is less instructive than the feedback received in supervised learning tasks (since the agent is not told the 'correct' action), but more instructive than in unsupervised tasks, where no feedback is given. For the remainder of this thesis, we will assume that all problem environments may be represented as finite state Markov decision processes (MDPs), which is a reasonable assumption within the GGP framework. Consequently, every possible game state may be described completely and uniquely by the values of a finite set of state variables.

In general, the values of state variables may be continuous, though they are most often either discrete (e.g. position of a piece on a board, value of a pixel), or may be treated as discrete by discretization of the variable's range. The set of all unique states made by combining different values of state variables is called the *state space*, and intelligently searching this potentially enormous set is a principal challenge in GGP and much of reinforcement

learning. For reasons described in Section 2.2.2, we will assume that all state variables take discrete values.

The MDP transitions between states when an agent takes some *action*, where the set of allowable actions may be determined by the current state. Upon taking an action and transitioning to some new state, the agent receives a reward. For many games that humans play, this reward signal is sparse, with 0 reward received at all state transitions but the final one. In GGP, this is always the case. Formally, we define an MDP as follows, making use of standard notation.

An MDP $M = (S, A, T, R, \gamma)$ is a tuple, where S is a finite set of states, A is a finite set of actions, $T(s, a, s')$ is the probability of transitioning from state $s \in S$ to state $s' \in S$ when action $a \in A$ is taken, $R(s, a, s')$ is the reward received for transitioning from s to s' with action a, and $\gamma \in (0, 1]$ is a discount factor reducing the value of future rewards. S must contain an initial state s_0 and at least one terminal state that ends the instance. An agent chooses actions according to a policy $\pi : S \times A \to \mathbb{R}$, which gives the probability of taking an action a in a given state s. Often, π will be deterministic, in which case a single action will have 100% probability for every state.

An optimal policy π^* is one that achieves the maximum possible cumulative reward when a terminal state is reached. Finding or approximating such a policy is the general goal of a reinforcement learner. A naïve (brute force) agent with knowledge of the MDP's dynamics might simply simulate taking all possible actions at the initial state, all possible actions at each of the resulting states, and carry on until only terminal states remain. This method of searching forms a tree that grows exponentially with the number of actions available at a state. For this reason, the size of the action set may be referred to as the *branching factor*. Exponential growth of the *search space* (i.e. the set of states simulated by an agent) causes brute force searching methods to become computationally intractable for problems of even modest complexity. Chess, for example, has an estimated 10^{50} possible states [5], which makes it infeasible to search all of them with current technology. Yet, computers can play chess well enough to defeat even the best human players [15] and have achieved similar results in games with even larger state spaces, such as Go [70]. This level of performance may be achieved by selectively expanding the search space in only the most promising directions. Some such methods are described in the following sections.

Depending on the nature of the agent, V or Q values may be computed, where $V^\pi(s)$ gives the value obtained by an agent at state s when following policy π and $Q^\pi(s, a)$ gives the value obtained by an agent that takes action a at state s and follows policy π thereafter. Formally, these values are given by the Bellman Equations:

$V^\pi(s) = \sum_a \pi(s, a) \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V^\pi(s')]$

$Q^\pi(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^\pi(s')]$

Extending this notation, $V^*(s)$ and $Q^*(s,a)$ refer to the optimal values for a given MDP, which are equal to $V^{\pi^*}(s)$ and $Q^{\pi^*}(s,a)$, respectively.

Gradient descent methods such as value or policy iteration may be used to find these tables, and through them, an optimal policy for an MDP. However, these are inappropriate for many RL tasks (including GGP), as they require knowledge of the dynamics of the MDP and are intractable in large state spaces. Instead, we focus on temporal difference and Monte Carlo methods.

2.1.1 Temporal Difference Learning

Unlike value and policy iteration, *Q-learning* and *SARSA* do not require the MDP's dynamics, nor do they need to update values for all possible states (including those that a reasonable policy might never reach). Instead, they learn from each action taken by taking the difference in Q-value of a future state and their current state. This quality is why they are called temporal difference (TD) methods. The key distinction between Q-learning and SARSA is in which future state is taken. Q-learning, which is off-policy, greedily takes the highest Q-value according to the following update rule.

$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s') + \gamma \max_a Q(s',a) - Q(s,a)]$

where α is a tunable learning rate. SARSA chooses the Q-value that corresponds to its current policy, according to:

$Q(s,a) \leftarrow Q(s,a) + \alpha[R(s,a,s') + \gamma Q(s',a') - Q(s,a)]$

Both methods are commonly used off-the-shelf RL techniques. However, they struggle in MDPs with extremely sparse reward functions. In GGP, rewards are only given in terminal states. For some games (e.g. Chess), this could be after tens or hundreds of moves with no reward signal at all. Monte Carlo Tree Search is better suited to these domains.

2.1.2 MCTS

Monte Carlo Tree Search (MCTS) is a lookahead method in which reachable states are represented as nodes in a tree. The current state forms the tree's root, states that are reachable with one action are its children, states that are reachable within two actions are their children, and so on. In addition to state information, each node contains two values.

The first counts the number of times that the node has been visited, and the second keeps track of the reward received on those visits. MCTS can be broken down into processes that are repeated, in order, until the state space is fully searched or a time/memory limit is reached.

1. **Selection**: Starting from the root, select a path of child nodes until a leaf node is reached. A measure of the 'goodness' of each child node can be obtained by looking at the average reward received when it was visited, which can be used to selectively exploit good nodes. However, exploration of nodes with relatively few visits is also necessary, which casts the selection process as a multi-armed bandit problem. UCB1 [7] is a commonly used solution, in which we select the node that has the highest value:

$$\text{UCB1} = \frac{r}{n} + c\sqrt{\frac{\ln N}{n}} \tag{2.1}$$

   where r is the total reward received from all visits to the node[1], n is the number of visits to the node, N is the number of visits to the parent node, and c is a tunable exploration parameter. When UCB1 is used with MCTS, the resulting algorithm is called the Upper Confidence Bound for Trees (UCT) method [41], and it is a common baseline in GGP.

2. **Expansion**: Unless the selected leaf node represents a terminal state, generate new child nodes for each state that can be reached from it. Choose one of them at random to simulate.

3. **Simulation/Rollout**: Produce a path from the selected child node to a terminal state by choosing actions randomly, or using some heuristic. Rollouts are frequent, and can be very long compared to the established Monte Carlo Tree (MCT), so it is important for them to be as lightweight as possible.

4. **Backpropagation**: Update r and n for every node on the path from the root to the terminal state reached.

Rollouts are the feature that gives MCTS an advantage in MDPs with sparse rewards. They allow an agent to see many terminal states without the investment of carefully

[1]For games in which two players take turns playing moves, we use the reward received by the agent currently acting.

analysing every turn along the path. As a result, MCTS is popular for use in GGP agents, as discussed in Section 2.2.1.

When an agent has run out of time for search, it must select an action to play. Typically, this is whichever action was the most visited during simulations, although it is possible to choose based on other criteria, like the greatest average reward. Unless otherwise stated, we will assume that the most visited action is selected. This decision is not a named part of MCTS, but we will refer to it as **Playing a move** or **Taking an action**.

It is possible to guide the search of MCTS by altering any of the enumerated steps above. We will discuss those related to the use of general heuristics in Section 2.2.4. Another relevant method for guiding MCTS is Memory-augmented Monte Carlo Tree Search (M-MCTS), which uses knowledge from similar states within the same game [95] to direct search. In it, a heuristic value is calculated and combined with the typical reward value during the selection phase. This heuristic value is calculated from similar game states that have already been encountered. While M-MCTS was originally limited to playing Go due to the use of a hand-crafted neural network architecture, it was later expanded to work with arbitrary games so that it could be applied to GGP [50]. Where these works address the problem of leveraging previously seen state information in the current game, we will seek to use information from states in a related, but different game.

2.1.3 State Abstraction

Although the methods described above aim to search the state space as efficiently as possible, it may still be beneficial to reduce its size by aggregating similar states and treating them as a single unit. This process is called *state abstraction*. A particular abstraction may be defined as $\phi : S \to S'$, where S is the original state space and S' is the reduced state space. In some cases, this may be done without any loss of information. For example, if the MDP is known to be a board game, or if a board can be identified at runtime [8, 44], symmetry may be exploited to group states that are effectively identical [70]. Other times, potentially useful information may be lost, but this is seen as a worthwhile trade-off for reduced complexity, even if it results in suboptimal choices some of the time. This is often seen for video games, where the value of a single pixel may be the difference between two states. It may therefore be desirable to aggregate closely related states by reducing pixel resolution [91] or by factoring the screen into regions [10].

Classes of abstraction scheme may be defined according to which properties of the MDP those abstractions preserve. In particular, classes ϕ_{Q^π} (preserves Q-values for $\langle policy, action, state \rangle$ tuples), ϕ_{Q^*} (preserves Q-values for optimal policies with $\langle action, state \rangle$

pairs), and ϕ_{a^*} (preserves Q-values for optimal policies and optimal actions at any state) all implicitly guarantee that an optimal policy in the reduced MDP is also an optimal policy in the original [49].

Unfortunately, such abstractions may still be costly to compute. In general, finding an optimal, smallest possible, abstract state space for a given problem is known to be NP-Hard [19]. Abel et al. [3, 2] developed algorithms for finding ϕ_{Q^*} abstractions in $\mathcal{O}(|S|^2)$ time, which is a great improvement, but still untenable for sufficiently large state spaces. To generate an abstraction scheme on the fly, we would like there to be no dependence on $|S|$ at all, which could mean that a guarantee of policy preservation is not possible. This leaves the door open for heuristics, such as the resolution reduction described above, to bridge the gap. For example, if ϕ_{Q^*} abstractions can be pre-computed for *source* games, then they may be applied to the target at runtime [93]. Lazaric [48] proposes a notion of *relevance* for determining sources from which to transfer, which may be estimated by taking a constant number of sample states from the target, and could therefore be relevant to GGP. Finding good state abstractions remains an active area of research in RL.

2.1.4 Reward Shaping

Reward shaping is a technique for guiding search within an MDP that has application to transfer learning in RL problems. The agent's perceived reward is modified to $R'(s, a, s') = R(s, a, s') + F(s, a, s')$, where R is the actual reward received and F is some shaping function. While any heuristic can be made into a shaping function, care must be taken to ensure that a policy found in the modified MDP remains useful in the original MDP of interest. In particular, Ng et al. [58] discovered *potential-based* shaping functions, which have the form $F(s, s') = \gamma * \Phi(s') - \Phi(s)$, where γ is the MDP's discount factor and Φ is a potential function depending only on the agent's state. In an analogy to the potential energies of the physical sciences, the value of F has no dependence on the path taken to get from s to s'. Since an agent must always begin in an initial state and finish in a terminal state (both of which may be assumed to be unique without loss of generality), the net effect of F on the cumulative reward received is the same for all policies. Ng et al. formalized this idea and proved that the use of a potential-based shaping function is a necessary and sufficient condition to guarantee that the optimal policy of the original MDP is preserved after reward shaping.

Other researchers generalized this result. Wiewiora et al. [94] showed that potential-based functions of the form $F(s, a, s', a') = \gamma * \Phi(s', a') - \Phi(s, a)$ also preserve the optimal policy, and Harutyunyan et al. [31] developed a method for learning a potential-based

approximation for *any* shaping function. This learning occurs in parallel with the main RL task.

The freedom to use any heuristic as a shaping function has allowed recent work to employ reward shaping as a mechanism for transfer learning between different RL tasks. Data from a source task (e.g. optimal policy, Q-values, etc.) may be used to shape the reward function of the target. Brys et al. [14] performed "policy transfer using reward shaping" (PTS) by treating the probability of an action being taken in the optimal policy of a source game (i.e. $\pi(s, a)$) as a shaping function using the conversion method established by Harutyunyan et al.[2] Experiments performed in a few sample RL tasks showed the performance of this method to be comparable to direct policy reuse (discussed further in Section 2.3.1). Although it did not always provide as much of an improvement in initial learning, it tended to be more robust to differences between the source and target tasks. Later methods augment this robustness through the use of ensembles of different shaping functions [13].

Reward shaping is an active area of research and an attractive option for RL domains that have otherwise sparse reward functions.

2.1.5 Lifelong Learning

Lifelong learning is an RL paradigm that is similar in purpose to GGP. An agent strives for the best performance in a series of MDPs drawn from a set, $\mathcal{M}$. Typically, MDPs in $\mathcal{M}$ share an action and state space, but differ in reward and transition probabilities. Although it is possible for an agent to face each new task with a blank slate, there is benefit in transferring knowledge from task to task over the "lifetime" of the agent.

Research in this area tends to focus on guarantees. In particular, a probably approximately correct (PAC) guarantee for MDPs (PAC-MDP) specifies that a policy's values are optimal within some error, ϵ, with high probability, $(1 - \delta)$ [75]. In GGP, where a shared state and action space are unlikely, a PAC-MDP guarantee is impossible, but this does not mean that lifelong learning research is useless for our purposes. Many concepts and tools are shared, such as state abstraction [3, 2, 93], policy transfer [12, 4], and value transfer [4]. These topics are each discussed in their own sections.

[2] Curiously, even though $\pi(s, a)$ could have been used directly as a potential-based shaping function, the approximated shaping function performed better empirically. The authors suggest that this may have been due to the magnitude of the shaping values, but do not perform a thorough investigation.

2.1.6 Deep Reinforcement Learning

Although it is not a focus of this work, deep reinforcement learning must be acknowledged as a driving force in the domain of game playing. Convolutional neural networks, in particular, have seen great success in board games and video games due to the importance of spatial information in these settings. Google's DeepMind team incorporated them into AlphaGo (and later, AlphaZero), a world-class Go bot, where they are used to guide MCTS, and may be trained with [70] or without [71] expert human knowledge. DeepMind have also used convolutional nets in combination with deep long short-term memory (LSTM) networks to achieve near world-class performance in StarCraft II [90], a video game with action and state spaces many orders of magnitude larger than those of Go [91]. State-of-the-art results have also been achieved for Atari games [57] and Dota 2 [16].

A key difference between learning single, ambitious games (i.e. 'grand challenges') and GGP is the time permitted for training. In the sphere of grand challenges, raw performance has historically been prioritized over training efficiency. For example, AlphaGo Zero was trained for 3 days and played through 4.9 million training games to achieve super-human performance at Go [71].

However, some more recent research has sought to maximize performance while sampling the environment a limited number of times, and by extension, limiting the amount of computation required for training. Kaiser et al. [38] set the Atari 100k benchmark, which limits training to 100,000 samples, approximately as many as a human would see in 2 hours of play. Schwarzer et al. [69] achieved super-human performance for 26 Atari games under these conditions, training for 10 hours on a single CPU and GPU. This state-of-the-art result shows that good learning can be achieved without requiring thousands of machine-hours of training. However, it is still too slow to be used in GGP at this time.

In other recent research, some success has been shown for zero-shot learning with AlphaZero-like architectures [72], which can be performed on time scales more appropriate to GGP. Although this work made use of Ludii, a different GGP framework than the one that we use, its goals of achieving transfer between similar games are very similar to our own.

2.2 General Game Playing

General Game Playing (GGP) emerged as a field of study in 1992, with Barney Pell's description of a "Metagame" [59]. Under this system, an agent was required to play any

game belonging to a class of two-player, zero-sum, perfect information, deterministic games played on a grid. This included games such as Chess, Checkers, and Tic-tac-toe.

Beginning in 2005, much of GGP research coalesced around Stanford's GGP framework and its associated annual GGP competition [28]. This research tended toward even more generality, with support for games that were single or multi-player, simultaneous or turn-taking, non zero-sum, and not confined to a grid. Although less commonly used, Stanford's framework has also been expanded to accommodate games that are non-deterministic or that give players imperfect information [62, 86].

Stanford's GGP environment provides a server (called the *Gamemaster*) that administers games for one or more GGP agents [52]. It is responsible for maintaining the game state and handling all communication. To begin a game, the Gamemaster provides all players with a specification of the game and a fixed period of time for initialization (typically on the order of tens of seconds, and up to a few minutes). During play, it requests moves from the agent(s) (which must be given in a few seconds), verifies that they are legal, updates the game state, and notifies the agent(s) of the changes. When a game ends, the Gamemaster terminates it and records the reward received by each player.

In GGP, reward can only be assigned when a game is finished. Under the framework we will be using, it must be an integer value in the range $[0, 100]$. For a two-player game with one winner and one loser, it is typical for the winner to receive a score of 100, for the loser to receive 0, and for both agents to receive 50 in the event of a draw. However, this does not have to be the case. A game could, for example, assign points based on the number of moves taken to win, or on the amount of material remaining to each player.

For compatibility with Stanford's framework, games must be encoded using the Game Description Language (GDL) [52]. There are now offshoots of GGP that use their own languages, such as VGDL for video games [66], or Ludii for ludeme-based game descriptions [61]. However, we will be primarily concerned with the original GDL.

2.2.1 General Game Playing Competitions

From 2005 to 2016, annual GGP competitions were held and co-located with either AAAI or IJCAI [27]. Like their own miniature conferences, they served as a hub for collaboration and dissemination of research for the GGP community, as well as a source of friendly competition. The main event focusses on two-player games so that competing agents could play against each other, one at a time, in a bracket structure similar to a professional sports tournament [28]. Over years of competition, no single team emerged as clearly dominant, though some won more than once. The most successful agent, CadiaPlayer (developed at

```
1:  (role xplayer) (role oplayer)
2:
3:  (<= (base (control ?p)) (role ?p))
4:
5:  (<= (next (control xplayer))
6:  (true (control oplayer)))
7:
8:  (<= (next (control oplayer))
9:  (true (control xplayer)))
10:
11:
```

```
1:  (<= (next (pedagogies gooseberries))
2:  (true (pedagogies entrepreneurial)))
3:
4:  (role gooseberries)
5:
6:  (<= (next (pedagogies entrepreneurial))
7:  (true (pedagogies gooseberries)))
8:
9:  (role entrepreneurial)
10:
11: (<= (base (pedagogies ?p)) (role ?p))
```

Figure 2.1: Example lines of GDL code from the game description of tic-tac-toe [29]. The original GDL code is shown on the left, and an obfuscated version is on the right. For the non-obfuscated code, line 1 designates *xplayer* and *oplayer* as valid player roles, line 3 defines *control* to be a base proposition that may be true when its parameter is a valid role, and lines 5-9 are rules that enforce alternating turns for the two players.

the CADIA research lab of Reykjavik University) won four times during this period. Each competition also hosted a human vs. machine exhibition event, in which a lone human representative played against the best artificial agent. From 2007 onward, humans lost consistently to the bots, as 2008 was the year that MCTS made its first appearance in competition [27].

More recently, (beginning around 2014 [79]), General Video Game Playing (GVGP) emerged as a field of study, and has held its own associated competitions. These competitions are divided into two tracks: one for planning agents that receive a coded game description in Video Game Description Language (VGDL) and low initialization time, and one for agents that receive raw pixel values and a much longer initialization time (on the order of hours) [88, 60]. The latter track is intended to allow deep learning methods to participate in GVGP, even though they cannot handle the short initialization characteristic of GGP. As a relatively new field of study, GVGP has many open problems, such as how to best parse visual input, and how to balance computation time with the need to act many times a second.

```
1: (succ 1 2)
2: (succ 2 3)
3: (succ 3 4)
4: (succ 4 5)
```

Figure 2.2: An example of a short numeric chain encoded in GDL. This pattern appears frequently to establish an ordinal set of symbols, such as the rows and columns of a board.

2.2.2 Game Description Language

Games played in the Stanford GGP system (and by extension, this thesis) are specified in Game Description Language (GDL), a variant of Datalog, which is itself a subset of Prolog [26]. A 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 [52]. GDL provides a set of built-in keywords that are common to all games. They are *role*, *base*, *init*, *true*, *does*, *next*, *legal*, *goal*, and *terminal* [29]. Together, they specify what constitutes the game state, what is true initially, which actions may be taken at which times, when the game ends, and what rewards are received by each agent when it does.

GDL also allows the user to define rules and relations that make use of game-specific constants, variables, functions, and predicates. Unlike the built-ins, these keywords will be subject to obfuscation (i.e. scrambling their names) at runtime. This grants agents insights into the structure of a game while denying semantic information through the particular vocabulary used. In addition, obfuscation randomizes the order of independent blocks of GDL code in order to confound simple attempts to recognize a familiar game. Figure 2.1 gives examples of non-obfuscated and matching obfuscated GDL code from the description of tic-tac-toe [29].

In general, GDL is very Spartan. There is no arithmetic; numbers are not differentiated from other symbols. Games that require functions like basic addition must define them. In practice, this means that game descriptions often include long sequences of numbers linked by a successor function, as in Figure 2.2. We call these patterns *numeric chains*. In GDL, they are necessary for imposing relationships like the order of rows and columns of a board, or the maximum number of possible turns before a game is ended. Especially in the latter case, it is not uncommon for these chains to link over 100 numeric symbols.

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

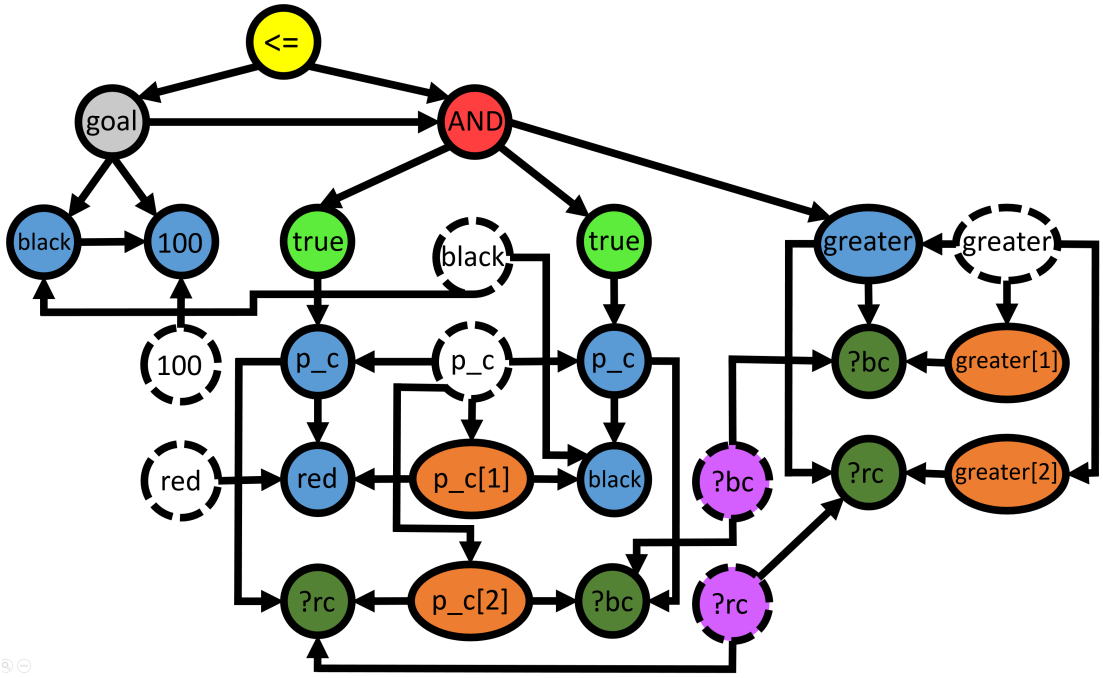

Figure 2.3: An example GDL statement from Checkers, in the GGP base game repository, and the rule graph generated from it. This code specifies that a reward of 100 is assigned to black when it has more pieces than red at the end of the game. Different colours indicate different types of nodes according to Genesereth's method [29]. A dashed line around a white background indicates a *symbol* node, which is important for mapping entities between two games.

Game	# Nodes	Time to Build Graph (ms)
8 Queens, Gd.	464	1.81 ±1.64
Tic-tac-toe	469	1.75 ±1.83
Connect Four	553	1.86 ±1.90
Rubik's Cube	1521	2.85 ±5.64
Checkers	3990	4.73 ±7.42
Chess	5668	7.88 ±13.30

Table 2.1: Rule graph sizes for games of varying complexity.

2.2.3 Rule Graphs

Stanford's GGP framework [64] includes code for parsing each GDL statement into a directed tree. This tree may be further refined into a *rule graph* using a method like those described by Genesereth [29] or Kuhlmann [46]. A rule graph is an unweighted directed graph where each node is assigned one of twenty-one colours. Each colour represents a logical connective (e.g. AND, OR), a GDL built-in (e.g. next, terminal), or a type of user-defined entity (e.g. predicate, variable). The original (obfuscated) names of user-defined entities 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[3]. Edges (with few exceptions) represent parent-child relationships in the GDL (e.g. the children of an AND node are its arguments). Figure 2.3 gives an example of the rule graph produced from a short segment of GDL code.

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 2.1 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 [45] proved that if two rule graphs are isomorphic, then the games that they represent are equivalent. Jiang et al. [35] 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 [45] extended it by looking for isomorphisms within a set of predetermined vari-

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

ations, using an off-the-shelf isomorphism solver. For example, they were able to identify games that were identical, but for different board sizes, numbers of pieces, or turn limits.

Of course, it is unreasonable to suppose that we could have a library of all possible rule graph variations from which to find isomorphisms. It would therefore be useful to have a criterion less strict than isomorphism. Soni and Singh [73] showed that it is possible to perform transfer using a homomorphism to map target to source task, but finding such a homomorphism is known to be an NP-complete problem [30]. Likewise, finding the largest isomorphic subgraph between two rule graphs could be viewed as a measure of similarity, but the subgraph isomorphism problem is also NP-complete [18]. In Chapter 3, we discuss a greedy method for measuring graph similarity in polynomial time.

2.2.4 General Heuristics

When it is too costly to fully explore the state space of a game (which we can assume to be true for all games complex enough to be interesting), a portion of the space may be searched more effectively by the application of heuristics. These heuristics hone in on features of a game state that are likely to contain useful information, like the state abstractions described in Section 2.1.3. A combination of heuristic values may be combined into a single number reflecting the perceived value of a state, in place of knowing its true value. When the game is known in advance, heuristics present an opportunity to incorporate prior game knowledge. In Chess, for example, human players commonly use the heuristic that one knight is worth about the same as one bishop or three pawns. However, the restrictions of GGP prevent the use of this kind of knowledge, as we do not know what a game's state will contain before playing it.

General heuristics are those that can be calculated for any game (or in our case, any game specified by GDL). Although they may not be as informative as hand-crafted, game-specific heuristics, general heuristics have been used successfully in numerous GGP agents [79]. At their most general, heuristics may be completely agnostic to the state of the game, because they are instead derived from the shape and features of the search tree, as in [25]. One such heuristic is mobility, which compares the number of moves available to an agent with the number of moves available to other players in the game. Mobility was used in Cluneplayer [17], the winner of the first International GGP Competition.

Some other heuristics require an initialization period during the game's start clock to extract information from the rule description, or from a number of (usually random) game simulations. These heuristics are general in the sense that they can produce an evaluation function for any game, even though the nature of that evaluation function will

vary from game to game. A relevant example is symbol counting [78]. In many games, it is advantageous to have more pieces than the opponent, which is likely to be reflected in the number of occurrences of some set of symbols in the game's state. However, a state will also typically contain symbols that offer no useful information when counted. A determination of the relative usefulness of the various symbols can be made during the start clock, and in [78], used 95% of that time.

Another relevant general heuristic requiring initialization is the history heuristic, which evaluates a possible move based on the reward that was received when playing that move in previous simulations [78, 80, 24].

If an agent can afford the cost of querying the state machine, approximate goal evaluation [32, 78] can be a viable heuristic. This method examines how close a state comes to satisfying a goal requirement (which would terminate the game with some known reward). In combination with methods for finding the number of steps needed to change the state in a particular way [55, 56], an agent may actively seek out a particular goal. Goal conditions may also be used to construct heuristics based on properties like their cardinality and domain [54].

In GGP, it is also possible to use a less general class of heuristics that must make assumptions about the nature of a game in order to function. Since board games are encountered frequently in GGP, there are methods for finding sequences of numbers, board coordinates [43, 68], and movable pieces [37], which can then be combined to produce heuristics, such as the distance on a board between two pieces. Although these heuristics may be very useful when they are applicable, their use does sacrifice some generality.

Once a set of general heuristics has been decided upon, further decisions must be made about how to weight and combine those heuristics, as well as how they will be leveraged to guide search. It is possible to hard-code a set of weights for heuristics in advance, but it is not generally true that the same heuristics will be equally useful across all games. For example, heuristics like mobility and symbol counting may be useful in Checkers, but are useless in Tic-Tac-Toe, where the players' decisions have no effect on either. Instead, we can try to determine appropriate weights during initialization, or while actively playing the game [67]. By performing an analysis of many games described in Ludii, [74] showed that it is possible to make reasonable guesses about the effectiveness of various heuristics based on the ludemes used in the description of the current game. This work is close in purpose to our own, as it allows an agent to shortcut the process of weighting different heuristics. However, it does not allow initialization to be skipped for those heuristics that require it.

A set of heuristics combined into one evaluation function can be used to effectively

guide search. Although we will be concerned with guiding MCTS, other forms of search may be used. In [53], general heuristics (including symbol counting) are used successfully as an evaluation function for minimax search. Walędzik and Mańdziuk [92] suggest several methods for guiding MCTS using a heuristic evaluation function, including modifying the UCB1 function, using it to guide rollouts instead of making a random selection, and terminating rollouts early by using the heuristic value instead of a true terminal reward value.

2.3 Transfer Learning

Although we may assume that it is not possible to comprehensively search the state space of a game at runtime, we may perform any amount of pre-computation that can be saved to disk. In particular, an agent may play other games (which are unlikely to exactly match our game of interest) for many iterations each. Knowledge gained from these games may be stored in the long term as a value table, Monte Carlo tree, or another structure appropriate to the agent's RL method. This knowledge forms a database of the agent's past experience, and may be drawn upon if the agent can identify analogous features of the game of interest at runtime. Doing so creates a matching of a *source* (archived) game to a *target* (new) game.

We may broadly classify transfer methods according to the kind of information that is transferred, which is often tied to the ways in which the source and target games are allowed to differ. When they are very similar (e.g. share a state and action space), it may be possible to use policy transfer methods. For games of a more distant or unknown relationship, it is necessary to find a mapping among states and actions from the source game to the target. Having such a mapping opens up new possibilities, as it enables the transfer of information like value functions and heuristics in cases where it would otherwise not be possible. These methods may be able to facilitate transfer between games that are less similar (e.g. different state/action spaces), but generally still require that at least the goal of the source and target games are closely related.

The success of a transfer learning method may be measured by comparing the performance of a transfer agent to an agent learning tabula rasa (i.e. from nothing). This comparison may be expressed through one of numerous evaluation metrics, including the *jumpstart* (initial performance gain), the eventual asymptotic performance, and the time taken to reach a particular performance threshold [85]. GGP also offers the opportunity to compare against agents using completely different strategies in a competition setting,

though this is more sport than evaluation, as there are many variables besides transfer learning to consider.

In the sections that follow, we first address policy transfer, then examine value and heuristic based methods that are more amenable to GGP. Although transfer learning is a prominent technique in deep learning [81], we omit a a discussion of its vast literature here. Instead, we focus on the methods most applicable to GGP.

2.3.1 Policy Transfer

Policy transfer is the process of reusing policies found in the source game directly in the target game. For example, Fernández and Veloso [23] balanced the exploitation of a library of past policies with random exploration of new ones in a maze navigation problem. This was possible because different tasks used the same maze and game dynamics, changing only the location of the goal. Even such an extreme similarity between source task and target is considered to be transfer learning, but is very unlikely to occur in a GGP setting. In a follow-up paper [22], this method was used to transfer knowledge from the domain of 3v2 robot keepaway to 4v3 robot keepaway[4]. Although these are still very closely related domains, the action space of the 4v3 game is larger, so a mapping between action spaces is required. For this work, the mapping was hand-coded and supplied to the agent at no cost. This approach was previously taken by Taylor and Stone [84], who applied policy transfer to SARSA in the same problem domain using a hand-coded action mapping. Both papers show a jumpstart in learning, but could not be applied directly to GGP, where the action mapping would have to be found autonomously.

In another example of policy transfer, Konidaris and Barto [42] relaxed source-target similarity restrictions by introducing the notion of 'agent-space', the space of sensory information received by the agent, which does not include all aspects of the full state space. Under their model, policies[5] may be transferred between agents occupying the same agent-space. In particular, policies for maze navigation may be transferred from one agent to another, provided that they are able to sense the distance to objects in the maze (in this case, doors and the keys that open them) in the same way. Notably, it was not required that the maze itself be the same.

[4]Robot keepaway, which appears as an experimental testbed frequently in the literature, is a game where simulated robot 'keepers' must keep a soccer ball away from 'takers'. In a 4v3 setting, there are 4 keepers and 3 takers.

[5]Strictly speaking, this work deals with *option* policies, which are constructed from macro-actions called options. An option is a short block of actions performed in sequence that may be discovered automatically. This builds on a substantial body of work (e.g. [76, 77, 89]) that is outside of our present scope.

Although this work is more amenable to the GGP setting than previous methods, the requirement that agent-space remain constant still poses a fundamental problem. Firstly, it cannot generally be assumed that it holds, but even if it does hold, the obfuscation of the target game's entity names makes that fact difficult to detect. It then becomes necessary to create a mapping for the entities of the source to those of the target, which leaves us with a method that requires a comparable amount of work to the heuristic/value transfer methods below, but may only be used on a much narrower set of source-target pairs. This brittleness also applies to the policy transfer methods discussed previously, and is a logical consequence of the kind of information being transferred. A policy is a prescription: a mapping $S \rightarrow A$. When the target game is different enough from the source that taking exactly the same action in all states is not very useful, there is no other information to fall back on.

2.3.2 Value Function/Heuristic Transfer

Unlike policy transfer, methods that transfer state values or other heuristics provide a notion of the 'goodness' of a state. They implicitly define a 'best' policy that the agent may choose to use as a starting point for learning in the target game, while also providing information about the quality of other policies. In the simplest case, it may be possible to directly copy Q-values [4], or learn a model from sample (s, a, s') tuples in the source and target games (e.g. [6, 11, 20]). However, both of these methods require the source and target domains to have very similar or shared state variables.

In order to handle less similar state spaces (or even identical state spaces that have had their variables renamed and reordered), it is necessary to represent the relationships between entities as a graph structure. Heuristics based on this structure commonly involve some method for finding a state/action mapping, although it is possible to do without. One example is given by Banerjee and Stone [9], who assigned heuristic values based on the shapes of 2-ply look-ahead trees, where nodes could be assigned values of 'win', 'loss', 'draw', or 'non-terminal'. Although successful for very short games, this approach does not scale to longer, more complex games, where a 2-ply look-ahead would be overwhelmed by non-terminal states, and the search cannot be made deeper without exponentially increasing the computation required.

The remainder of the methods we examine in this section all make use of at least a partial mapping between source and target games. Early works often relied on finding specific, hard-coded structures in the rules of a game. For example, Kuhlmann and Stone [43] found structures consistent with a board and pieces, and generated heuristics such

as the distance between pieces. Banerjee et al. [8] coded common concepts like 'making a line of pieces', and 'blocking the opponent' to transfer knowledge between tic-tac-toe, Connect-3, and a miniature version of Go. Taylor et al. [82] employed heuristics based on the configuration of nearby walls for maze tasks. While these approaches are useful for transfer between certain types of games, their scope is quite narrow because they take advantage of particular game tropes. For example, many of the games that humans have invented employ pieces and boards, but there is no reason that an arbitrary MDP should do so.

For small enough state spaces, it is possible to try all combinations of state/action mappings and take the best of them (as in the MASTER method[83]), but this approach is computationally untenable, in general. The amount of work can be reduced somewhat by starting from individual state/action pairings and merging only those that are consistent into larger sets. Liu and Stone [51] took this approach by building on the Structure-Mapping Engine (SME), which is an implementation of the structure-mapping theory of analogical processing [21]. SME builds matchings from pairs of concepts that are semantically analogous (e.g. water flows downhill as heat flows to lower temperature). Liu and Stone adapted this method to the RL setting by assuming that a qualitative dynamic Bayes network (QDBN) existed for both the source and target game (where 'qualitative' refers to node and edge labels like 'discrete variable'), and forming matchings between their entities. This process was successful in producing good mappings for different numbers of players in the domain of robot keepaway. However, depth-first searching to merge consistent matchings is still quite computationally expensive, and the QDBNs had to be supplied by hand.

Chapter 3

Mapping

As described in Section 2.2.2, obfuscation of the GDL description of a game poses a significant barrier to transfer, since we cannot simply identify important features (like pieces, a board, scores, or actions) by name. Such features are likely to be shared by similar games, and we depend on this fact to transfer knowledge from the source game (which we will call G_1) to the target game (G_2). For example, we might know that taking a particular action in a particular state of G_1 leads to good outcomes, but even if we were to encounter exactly the same situation in G_2, we would not recognize it without first decoding the obfuscation. This begs a question: If game obfuscation is an artificial problem whose effects we intend to nullify, why insist on using it at all?

In describing a game for human players, it is useful to use common language that allow those players to draw analogy with other games. For example, telling a player that, 'These are your pieces,' conveys many ideas in few words. 'You will move these.' 'Their positions are important.' 'Losing them reduces your set of available moves.' However, none of this guaranteed by use of the word 'piece', nor prohibited if some other word were used instead. These real characteristics of the game are determined by its dynamics, as codified by its full set of rules, not the particular names that we assign to its entities. Checkers is still Checkers when the rulebook is written in another language.

Complete obfuscation of a game description takes this idea to its logical extreme, replacing the name of every entity with a new, random one. It forces us to disregard the semantic information associated with named entities in the game description to focus solely on the dynamics. This is the standard for GGP, and will be assumed for the entirety of this thesis. Of course, there would not necessarily be anything wrong with research that did make use of semantic information, since the rules for games that humans care about

(outside of GGP) are generally written in a way that are meant to be understandable. However, even if we were willing to make this assumption, it would still be necessary to verify the similarity among entities between two games. For example, both Checkers and Poker have 'kings', but those do not actually represent similar things.

In this chapter, we discuss the algorithms that form a mapping from the elements of a source game to those of a target game, which is a prerequisite for conducting transfer in later chapters. This work was published at the 2021 Conference on Games [36]. 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 chapter 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.

3.1 Background

3.1.1 Rule Graphs

As previously described in Section 2.2.3, a rule graph is a structure that captures all of the information present in the GDL description of a game's rules. Its nodes represent predicates, symbols, or built-in functions needed by the Game Master, and its directed edges (generally) indicate that one node is an argument to the other. While there are rare exceptions to this meaning of an edge, there is no ambiguity when the colour of both the parent and child node are known. For our purposes, this is what matters, since we will be

interested in the number of edges from one node to its neighbours of a particular colour. We will never be counting neighbours of all colours and treating those connections as equal.

In the work that follows, we will make use of the fact that if two rule graphs are isomorphic, then the games that they represent are equivalent [45], as well as the assumption that if two rule graphs are similar, then the games that they represent are likely to be similar. (We will also see examples where this assumption may be broken, in later chapters.) As such, we will need to quickly construct a rule graph for both the source and target game, and use the method described by Genesereth [29], which deterministically produces a unique graph, G, for each GDL description. For convenience, we now list the (paraphrased) steps of this method, with a small modification **in bold**.

1. Add a node to G for each occurrence (i.e. each individual appearance in the GDL) of a logical connective, predicate symbol, function symbol, and variable. If one occurrence is an argument to another, draw an edge from the latter to the former.

2. For each variable or symbol that occurs in the GDL description, but is not a GDL keyword add one node to G and draw an edge from it to all of its corresponding occurrence nodes constructed in step 1. (We will later refer to these new additions as *symbol nodes*.)

3. For each node, v, created in step 2 that corresponds to a symbol that takes arguments, create one node representing the position of each argument, and draw an edge from the symbol node to its argument nodes. (E.g. A predicate that takes two arguments would generate two argument nodes, one corresponding to its first argument, and one to its second.) Add an edge from each argument node to the corresponding occurrence nodes created in step 1. If v represents entailment or a binary GDL keyword, draw an edge from its first argument node to it second.

4. Assign a colour to each node to indicate its type, where each logical connective and GDL keyword have a unique colour, and all user-defined symbols are labelled as either a predicate occurrence, function occurrence, variable occurrence, argument, variable symbol, or non-variable symbol. **At any later time, a subtype may be assigned to non-variable symbols. When a subtype is known, it functions as the node's colour.** (Subtypes are not used in this chapter, but will be important to the handling of number chains in the next.)

For an example of a GDL code excerpt converted to a rule graph, see Figure 2.3, and for examples of rule graph sizes resulting from games of varying complexity, see Table 2.1.

3.1.2 Related Work in Mapping and Transfer

Although we are not aware of any other works that have performed symbol mapping for the purpose of transfer in GGP, there has been some research in mapping specific kinds of games. Falkenhainer et al. performed transfer between non-identical games by extending the Structure Mapping Engine [21]. Klenk and Forbus [40], were able to establish mappings between kinematics problems, while Hinrichs and Forbus [33] 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 [43, 68], and moveable pieces [37], which can then be combined to produce heuristics. We will examine and make use of some of these features in later chapters. Game-independent features may also be discovered autonomously [39], and whole games may be decomposed if they are made up of smaller sub-games [34].

Section 2.3 gives several examples successful transfer in games, when a mapping has been given in advance. Methods like these may be used for transfer after running our mapping algorithm.

3.2 Method

3.2.1 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 [29] (outlined in Section 3.1.1) because, unlike Kuhlmann's method [46], it does not require coloured edges, which makes processing the graph somewhat simpler. Not including subtypes, there are 24 possible node colours, among which a few are particularly important for our purposes. We now draw attention to them, their meanings, and the structure of their graph neighbourhoods.

In Figure 3.3, *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.

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.

3.2.2 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:

$$\frac{\sum_{neighbours} distance}{(\#\ of\ neighbours) + 1} \tag{3.1}$$

where neighbours are 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

```
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
```

Figure 3.1: Algorithm for finding the distance of two rule graph nodes (Part 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.

```
1:  function ListDistance(nList_1, nList_2, depth)
2:      for each node ∈ nList_1 or nList_2 do
3:          if node if it would form a cycle then
4:              remove node from its list
5:          else if node.colour is NON_VAR_OCC AND depth > 1 then
6:              replace node with its parent SYMBOL node
7:      assigned, tuples ← ∅
8:      totalDist, count ← 0
9:      for each n_1 ∈ nList_1 do
10:         for each n_2 ∈ nList_2 do
11:             dist ← Distance(n_1, n_2, depth)
12:             if dist = 0 then
13:                 add n_1 and n_2 to assigned
14:                 remove n_1 from nList_1 and n_2 from nList_2
15:                 count ← count + 1
16:             else
17:                 add (dist, n_1, n_2) to tuples
18:     sort tuples in increasing order by dist values
19:     while not all nodes ∈ assigned AND tuples is not empty do
20:         (dist, n_1, n_2) ← first element popped from tuples
21:         if n_1 ∉ assigned & n_2 ∉ assigned then
22:             add n_1 and n_2 to assigned
23:             remove n_1 from nList_1 and n_2 from nList_2
24:             totalDist ← totalDist + dist
25:             count ← count + 1
26:     for each remaining node ∈ nList_1 or nList_2 do
27:         totalDist ← totalDist + 1
28:         count ← count + 1
29:     return totalDist, count
```

Figure 3.2: Algorithm for finding the distance of two rule graph nodes (Part 2).

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 Figures 3.1 and 3.2. 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 3.3. 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 (Figure 3.1) ensures that symbol

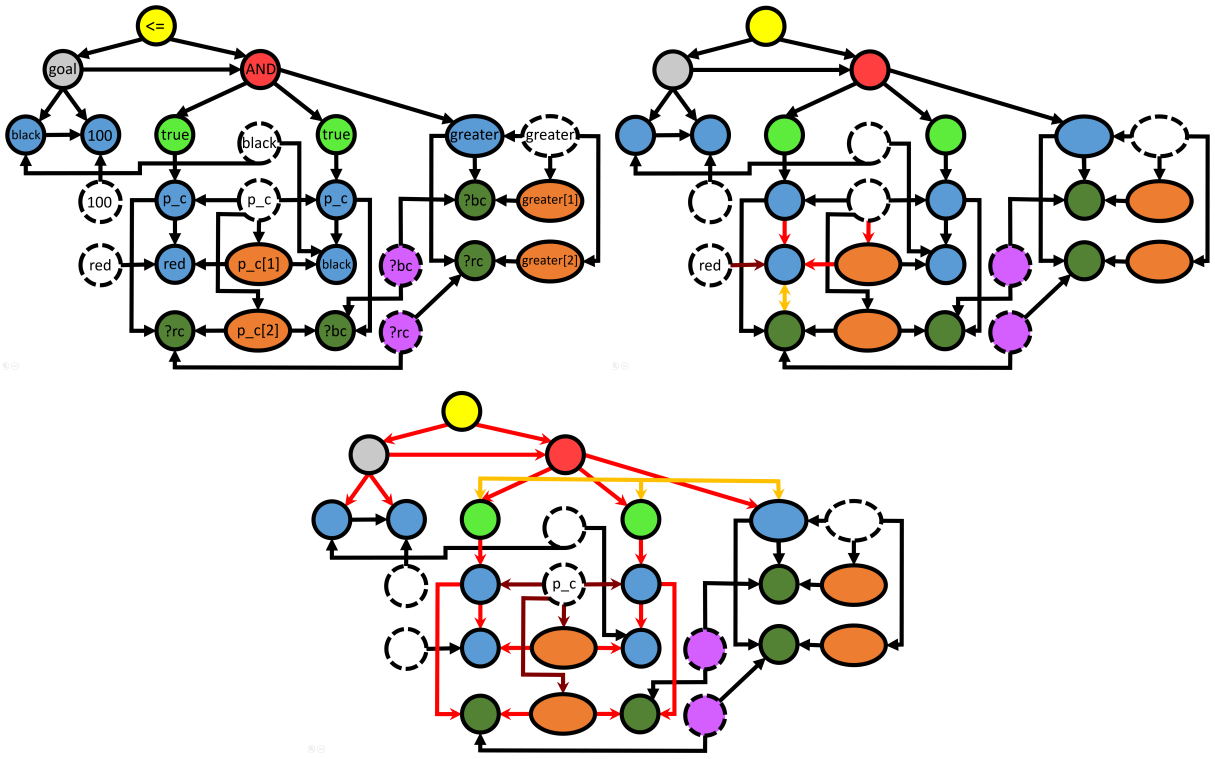

Figure 3.3: (Top Left) The rule graph generated from the GDL code in Figure 2.1. Names are overlaid for clarity, although they are not normally visible. Symbol nodes have dashed edges (variable symbols are purple, non-variables are white). Non-variable occurrence nodes are blue. Variable occurrences are dark green. Argument nodes are orange. Each built-in GDL keyword has a unique colour. (Top Right) 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. (Bottom) Edges discovered, starting from 'piece_count' (p_c).

nodes can only be expanded at $depth = 0$ (i.e. initially). Argument nodes may only be fully expanded at $depth <= 1$ (Figure 3.1, line 8); otherwise, we only check which symbol is its parent. Occurrence nodes are replaced by their corresponding symbol node if $depth > 1$ (Figure 3.2, line 5), 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 (Figure 3.1) enforces a maximum search depth. From line 6 (Figure 3.1), variable occurrence nodes are never expanded. Line 3 (Figure 3.2) prevents cycles. From lines 9 to 25 (Figure 3.2), 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 12-15 (Figure 3.2) represent an optimization that allows a pair to be assigned early, if it is a perfect match. Lines 26-28 (Figure 3.2) assign a maximum distance of 1 to any nodes that were not matched.

3.2.3 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:

$$DS(n_1, n_2) = \alpha_D(\frac{Dist(n_1, n_2)}{D_{max}}) + \alpha_N(1 - \frac{Num(n_1, n_2)}{N_{max}}) + \alpha_A(1 - \frac{Assign(n_1, n_2)}{A_{max}}) \tag{3.2}$$

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.

3.2.4 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[1]. LMap employs the heuristic of Riesen et al.'s Greedy-GED algorithm [63], 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 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.

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

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.

3.3 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[2]"(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 [64]. These two games were chosen because they produce very differently sized rule graphs (given in Table 3.1), 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 [43, 68], and they often significantly outnumber all other nodes. Checkers, for instance, defines 202 numeric

[2]"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.

symbols, compared to only 50 other non-variable symbols, which we are more interested in mapping.

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 [63].

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 3.4 discusses results, and additionally notes several informal experiments that cannot be readily evaluated.

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.

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[3] because they provide essential

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

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 that can actually be played. 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.

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 3.4.1.

3.4 Results and Discussion

3.4.1 Main Results

Self-Mapping

Table 3.1 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.

Game	# Nodes	Time to Build Graph (ms)	Myopic Self-Mapping % Correct	Time (ms)
8 Queens, Gd.	464	1.81 ±1.64	80.63 ±9.25	5 ±7
Tic-tac-toe	469	1.75 ±1.83	85.33 ±10.24	3 ±3
Connect Four	553	1.86 ±1.90	96.79 ±3.55	2 ±3
Rubik's Cube	1521	2.85 ±5.64	50.31 ±5.29	11 ±9
Checkers	3990	4.73 ±7.42	67.10 ±5.92	49 ±27
Chess	5668	7.88 ±13.30	65.71 ±4.20	57 ±26

Game	MMap Self-Mapping % Correct	Time (ms)	LMap Self-Mapping % Correct	Time (ms)
8 Queens, Gd.	**100.00** ±0.00	224 ±140	**100.00** ±0.00	59 ±28
Tic-tac-toe	**88.00** ±10.24	133 ±70	85.33 ±10.24	46 ±24
Connect Four	**100.00** ±0.00	125 ±60	**100.00** ±0.00	54 ±27
Rubik's Cube	**100.00** ±0.00	8612 ±451	**100.00** ±0.00	535 ±238
Checkers	**100.00** ±0.00	43680 ±4166	**100.00** ±0.00	1371 ±316
Chess	**97.14** ±1.21	86964 ±4400	95.18 ±2.34	3612 ±323

Table 3.1: Self-mapping results for games of varying complexity. The highest mapping accuracy for each game is in bold.

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.

Adding or Removing Nodes

From Figure 3.4, 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.

Mapping to Game Variants

Table 3.2 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.

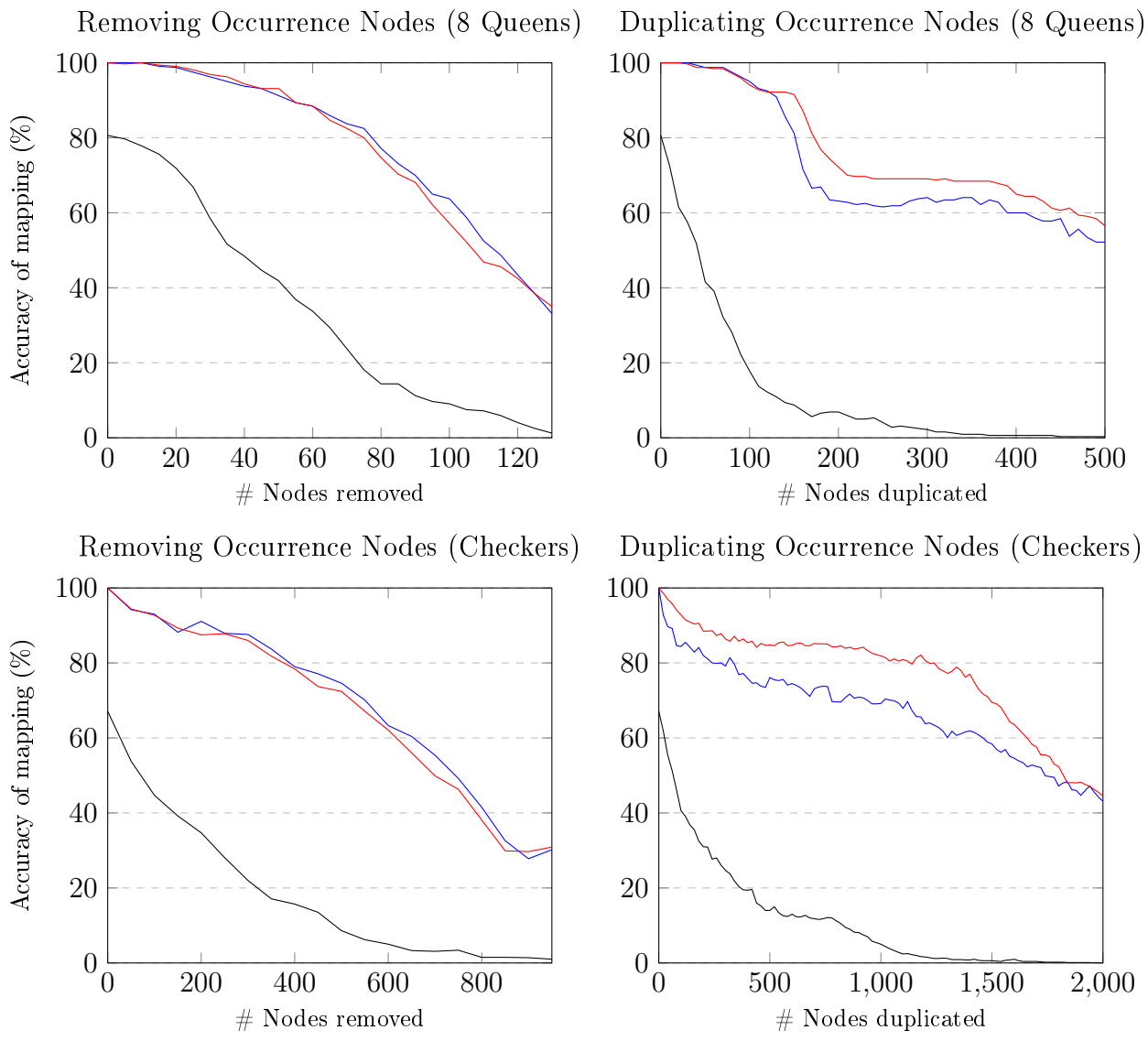

Figure 3.4: Mapping accuracy for alterations of 8 Queens (top) and Checkers (bottom). MMap in red, LMap in blue, Myopic in black.

Game Variant	% Sym. Shared	Myopic % Correct	Myopic Time (ms)
6 Queens, Unguided	93.33	89.29 ±8.60	5 ±8
8 Queens, Unguided	93.33	89.29 ±8.60	3 ±4
12 Queens, Unguided	93.33	75.71 ±6.55	4 ±5
16 Queens, Unguided	93.33	75.71 ±6.55	4 ±5
31 Queens, Guided	100.00	67.50 ±8.05	6 ±8
Checkers, Small (6X8)	97.96	61.15 ±4.38	45 ±28
Checkers, Tiny (4X8)	95.83	69.67 ±5.43	36 ±14
Checkers, Must-Jump	84.62	41.25 ±4.15	22 ±14
Checkers, Cylinder, Must-Jump	84.62	40.00 ±3.25	24 ±14
Checkers, Torus, Must-Jump	84.62	40.00 ±3.25	24 ±14

Game Variant	MMap % Correct	MMap Time (ms)	LMap % Correct	LMap Time (ms)
6 Queens, Unguided	**100.00** ±0.00	166 ±95	**100.00** ±0.00	54 ±24
8 Queens, Unguided	**100.00** ±0.00	203 ±110	**100.00** ±0.00	41 ±17
12 Queens, Unguided	**100.00** ±0.00	256 ±142	**100.00** ±0.00	62 ±34
16 Queens, Unguided	**100.00** ±0.00	300 ±96	**100.00** ±0.00	56 ±30
31 Queens, Guided	**87.50** ±0.00	584 ±197	68.50 ±0.00	123 ±80
Checkers, Small (6X8)	**97.92** ±0.00	36586 ±674	**97.92** ±0.00	1279 ±231
Checkers, Tiny (4X8)	**97.83** ±0.00	35996 ±640	**97.83** ±0.00	1214 ±210
Checkers, Must-Jump	**100.00** ±0.00	36446 ±280	86.02 ±4.84	2251 ±211
Checkers, Cylinder, Must-Jump	**100.00** ±0.00	34373 ±525	81.82 ±7.91	2351 ±172
Checkers, Torus, Must-Jump	**100.00** ±0.00	34618 ±557	80.80 ±6.95	2334 ±221

Table 3.2: Mapping results for the 8 Queens and Checkers families. The highest mapping accuracy for each game is in bold.

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. In the next chapter, we examine if LMap's mapping accuracy is sufficient to allow good transfer in a GGP bot.

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

Table 3.3: Mean distances from standard checkers.

A Note on Distant Mappings

We have not done a comprehensive analysis of mappings between more distantly related games, but this is an interesting direction for future work. 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.

Distance as Game Similarity

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 3.3. 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. This idea is explored further in Section 5.1.

3.5 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, 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.

Chapter 4

Transfer via General Heuristics

With the methods developed in the last chapter, we are able to identify similar symbols between different, but related games. These symbols include important features like actions and state variables. When combined with information like which states and actions resulted in good outcomes in a previously played game (the **source** game), this, in principle, allows us to identify states and actions in a new game (the **target** game) that are also likely to be good. However, our mapping algorithms offer no guidance on how best to go about this, as their output can as easily be used to guide search algorithms as they can temporal difference methods, or perform machine learning.

The short time limitations imposed by GGP constrain our choice, so for direction, we look to the GGP literature, where many papers and competitions have shown Monte Carlo Tree Search (MCTS) to be an effective method for achieving good play on short notice [79]. In standard MCTS, an agent begins as a tabula rasa, but through many fast simulations of a game, is able to estimate the quality of game states to guide its play. This approach, however, fails to capitalize on an agent's previous experience. Where a human might recognize, for example, that a game is just Checkers on a larger board, an MCTS agent must treat it as an entirely new game, even if it has previously played a regular game of Checkers. MCTS cannot normally analogize, so using it as a basis for our transfer agents is desirable for its novelty, as well as MCTS' well established position in the existing literature.

We augment MCTS with knowledge transfer via the use of general heuristics (heuristics that can be applied to any game in GGP). Although the use of such heuristics is not new, our method of initializing them is. In particular, we initialize a set of general heuristics by transferring data from the source game, enabled by the mapping algorithms of the last

chapter. These heuristics are then used to guide MCTS, resulting in a Transfer-Initialized Guided MCTS (TI-GMCTS) agent.

We compare the performance of this agent to one using the same heuristics, but initialized ordinarily via unguided simulations during the game's start clock. We refer to this as a Simulation-Initialized Guided MCTS (SI-GMCTS) agent. Our experiments show that TI-GMCTS is able to effectively transfer information across domains, where rule changes have modified the state and/or action space of a game. We show that TI-GMCTS compares particularly favourably to SI-GMCTS when initialization time is short.

In the following section of this chapter, we give additional necessary background. We then describe in detail how to calculate each of the general heuristics we have used, and describe a method for combining them into one evaluation function, weighted by their relative strength. We show how these heuristics are initialized, either through transfer or simulation, and we compare the performance of TI-GMCTS and SI-GMCTS agents in six variants of Checkers and five variants of Breakthrough with varying state/action spaces. We conclude by analyzing the strengths and weaknesses of a Transfer-Initialized agent, and suggest methods for expanding its capabilities in the future.

4.1 Background

This chapter requires background in MCTS and general heuristics, discussed in Sections 2.1.2 and 2.2.4, respectively.

Additionally, in order to choose a set of game variants from the infinitely many that are possible, we will refer to those variations described by Kuhlmann and Stone [45], which were identified by checking the rule graph for isomorphism with a pre-defined set of templates, in that work. For convenience, we list the possible variations here:

- **Number of pieces**: Pieces are added or removed from the initial state.
- **Configuration of pieces**: The position of pieces in the initial state is altered.
- **Board size**: One or both dimensions of a two-dimensional board are made larger or smaller.
- **Board topography**: The playing surface is swapped from a flat plane to a cylinder or torus.
- **Step limit**: The maximum number of turns before a game terminates is changed.

- **Inverted goal**: Rewards are swapped so that a winning position becomes losing, and vice versa.
- **Switched role**: In a game with two players, those players change roles. For example, Red to Black in Checkers.[1]
- **Missing feature**: A rule or other feature of the game is removed completely.

4.2 Method

4.2.1 General Heuristic Calculation

The three general heuristics described below and used in our experiments are not novel, and this chapter is not meant to be an examination of their effectiveness across a suite of games. Rather, we wish to show that when an agent has access to general heuristics that are effective, it can benefit more from them when they are weighted and initialized via transfer from another game. We have therefore chosen three heuristics from the literature (which are summarized in more detail in Section 2.2.4) that can be reasonably effective on Checkers and Breakthrough, our experimental test beds. There exist numerous variations on each of them in the literature, but we have based the versions below on [78] (although they are ultimately applied to MCTS in a way that is different from that work).

When transfer is not used, we initialize these heuristics with data gathered from regular UCT simulations for SI-GMCTS. These are performed for as much of the start clock as possible, and afterwards, they form the initial Monte Carlo Tree. In the descriptions below, we assume that transfer is not being used. Initialization with transfer will be discussed separately after all of our heuristics have been defined.

Mobility Heuristic For a two-player game, we define an agent's mobility in a given state to be the difference between the number of different possible moves that it can make, and the number of different possible moves that its opponent can make. (i.e. A positive mobility means that an agent has more available moves than its opponent.)

For each simulation run during initialization, we find the average mobility score across all states visited, and note the final reward value obtained. At the end of the start clock, we

[1] We do not actually consider this to be a variant, since we always alternate between roles in a two-player game. However, it is included for completeness with regard to Kuhlmann's work.

do linear regression on average mobility vs. reward, and also find the Pearson correlation coefficient (the r-value). Thereafter, we can find an expected reward for any given mobility value, V_{mob}, and weight it against other heuristics using the r-value ($W_{mob} = |r|$).

Symbol Counting Heuristic The Symbol Counting heuristic is one heuristic composed of many smaller heuristics that must be discovered during initialization. A game state is composed of some number of facts that are true. For example, a state in checkers might contain `(cell c 2 bp)`, indicating that there is a black piece in the second rank of the C file. If we were to count the total number of facts that contain `bp` as the third argument to `cell`, we would know the number of black pieces remaining, which is an intuitive heuristic for Checkers. However, an agent does not know which symbols are important when first given the game description.

To discover them, it tracks the number of all symbols in each position at which they appear as arguments in each type of fact (e.g. `cell`). In other words, there is a count associated with each *(symbol, fact, position)* triple. Generally, this produces many triples which provide no meaningful insights into the quality of a game state. Some of these may be filtered out by observing which symbols are unaffected by player actions. If there is no variance in the count of a triple across all simulations, then it is discarded as a heuristic. For example, every game state in checkers contains a fact like `(cell c 2 X)`, where `X` may indicate a black piece, red piece, or empty space. Triples involving the `c` or `2` symbols will be discarded.

For the remaining triples, we must determine how well each one indicates the value of a game state[2]. As with the mobility heuristic, we do linear regression for each triple at the end of the start clock, comparing its average count to the final reward received for each simulation. Then, a combined heuristic value for all symbol counts of a game state, V_{sc}, is given by the weighted average:

$$V_{sc} = \sum_{t \in T} \frac{V_t * |r_t|}{\sum |r_t|} \tag{4.1}$$

where V_t is the predicted reward for triple t, r_t is the Pearson correlation coefficient for t, and T is the set of all triples that appear in the game state, and were not discarded. We take the weight of this combined heuristic to be $W_{sc} = \max_{t \in T} |r_t|$.

History Heuristic Given a complete action, like `(move bp g 6 h 5)` (which specifies moving a black piece from G6 to H5 on the board), the history heuristic provides an

[2]Up to this point, we have employed the method for symbol counting described by [78]. Our method for combining the various symbol counts into one heuristic value differs.

estimate of its value by referring to previous simulations in which that action was taken. During initialization, every move played during a simulation is added to a set, and at the end of the simulation, the average reward of each of those moves is updated by the final reward received. Using the version of this heuristic specified by [78], it does not matter how many times a move was played during a simulation, only that it was played at least once. Since the presence or absence of each move is then a binary feature, it does not make sense to perform linear regression, as we have done for the other heuristics. Instead, the heuristic value, V_{hist}, is just the stored average reward. To assign a weight, W_{hist}, we assume that a value is more informative the further it is from the reward received for a draw [3]:

$$W_{hist} = \frac{2 * |V_{hist} - R_{draw}|}{R_{max} - R_{min}} \tag{4.2}$$

where R_{draw} is the reward received for a draw, R_{max} is the maximum possible reward received, and R_{min} is the minimum possible reward received.

In addition to this history heuristic dealing with very specific actions (e.g. moving a piece between two specific board coordinates), we introduce a general history heuristic that considers all actions of a type to be the same (e.g. all actions called `move`). This heuristic is only useful for games in which there are at least two different types of actions, and it is possible to run a complete simulation without including all of them. In the GGP base repository, Checkers is an example of such a game, since double- and triple-jumps are considered a separate action from other moves. A value (V_{gen_hist}) and weight (W_{gen_hist}) for each action type are calculated during initialization in the same way they are calculated for complete actions using the standard history heuristic.

Evaluation Function With all of our heuristic values and weights defined, we take their weighted average to get a complete heuristic evaluation function, V:

$$V = \sum_{h \in H} \frac{V_h * W_h}{\sum W_h}, H = \{mob, sc, hist, gen\_hist\} \tag{4.3}$$

This function is scaled to the range $[R_{min}, R_{max}]$, and may be used to estimate a state's value in order to guide search.

[3] We use language associated with two-player games here, but this definition may be expanded to include one- or more-player games replacing R_{draw} with another appropriate value, like the overall average reward across all simulations.

4.2.2 Guided Monte Carlo Tree Search

There are several possible options for using a heuristic function to guide MCTS [92], including biasing the selection phase, performing non-random rollouts, and terminating rollouts early with reward equal to the function value. Although our combined heuristic evaluation function is compatible with any of these methods, we have opted to focus on only the first. It is much less computationally expensive than heuristically guiding rollouts, since rollouts encounter many new nodes each simulation, and it is a much smaller commitment than early rollout termination, since reward is still based on true terminal states.

Figure 4.1 shows the number of rollouts completed over time in a game of Checkers by a predecessor of the transfer agents presented in this thesis. (It is discussed in more detail in Section 4.2.6.) We can observe from the figure that when rollout guidance is used (which is the case for the SR-agent), the number of rollouts completed is reduced to about half the number that may be completed when it is not used (the S-agent). On the other hand, using only selection guidance slows the agent only a little compared to UCT.

Compared to these older agents, it is even more untenable to perform rollout guidance in TI-GMCTS, because the mobility heuristic (which was not present in the older agents) requires querying the state machine for the possible moves in a state. This is much too computationally expensive to be worthwhile for nodes encountered during a rollout that will probably never be visited again. As a result, we made the decision to proceed with selection guidance only, moving forward.[4]

In order to influence the selection phase, we modify the UCB1 score that will be maximized:

$$\text{UCB1}_{Heur} = \alpha^n V + (1 - \alpha^n)\frac{r}{n} + c\sqrt{\frac{\ln N}{n}} \tag{4.4}$$

where $\alpha \in [0, 1]$ is a decay parameter , V is our heuristic evaluation function, and the remaining parameters are the same as in Equation 2.1. The use of a decay parameter on our heuristic value allows the accumulated reward to grow in importance as more visits are logged and it becomes a better approximation of the true value of the state.

4.2.3 Archived Information

Up to this point, we have have been concerned with an SI-GMCTS agent that is initialized by simulations performed during the start clock. We will now shift our focus to the

[4] We also did some experimentation with early rollout termination using the heuristic evaluation function, but were not able to produce performance comparable to the other agents.

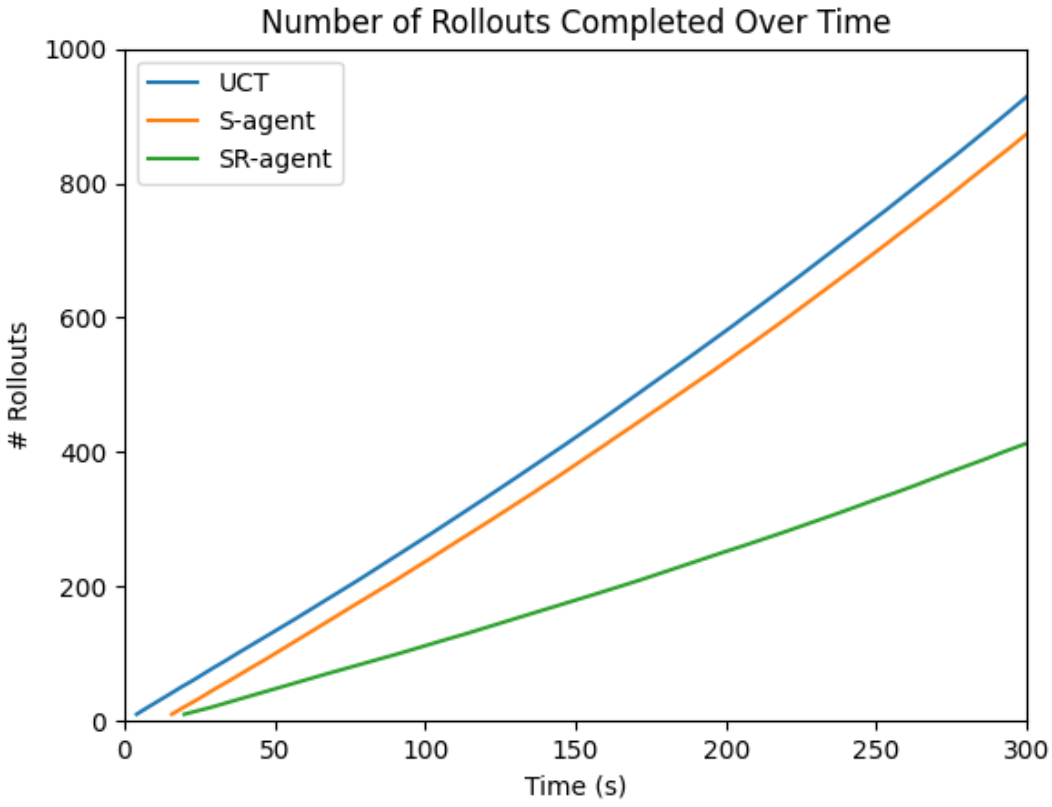

Figure 4.1: Number of rollouts completed over time by UCT, a selection-only transfer agent (S-agent), and a selection-and-rollout transfer agent (SR-agent). Values are averaged over 20 games.

additional infrastructure needed to support transfer for a TI-GMCTS agent.

Before we play a target game, it is necessary to construct an archive of source game data and pre-compute heuristic data from it. We begin by assigning two UCT agents to play a number of iterations of the source game against each other, while logging all of the data that an SI-GMCTS agent would log during its initialization period. We use UCT rather than heuristic-guided agents for this purpose to avoid biasing the path that a game takes. When referring to a UCT agent, we will assume an MCTS agent guided by UCB1 [7] with an exploration parameter of $\sqrt{2}$ and a discounting factor of 0.98.

These agents continue logging even after they have begun playing moves, and will not stop until the game ends. Compared to SI-GMCTS, this results in a significantly larger number of simulations to draw upon after a single game, and the states discovered in those simulations are more closely grouped around the path through the state tree that the game actually took. Any number of games may be archived in this way. In this work, we have used 100. For a full, actively competing TI-GMCTS agent, this archive would be expanded with every game that the agent plays.

We do not want to process all of this raw data during the start clock, so relevant heuristic parameters are pre-computed offline. For the mobility and symbol counting heuristics, this means doing linear regression across the data stored for all games, and writing the resulting slopes, intercepts, and r-values to file. For the simpler history heuristics, we keep a running total of reward received to produce a final average across all games. The final result is a short file that can be read in < 30 ms to initialize all of our heuristic parameters. More detail is given in Section 4.2.5.

Along with the heuristic data, we store a file detailing the rule graph structure of the source game to be used for symbol mapping. Since this structure is the same every time the source game is played, we need only store one copy of it.

4.2.4 Rule Graphs and Symbol Mapping

We begin the initialization period by generating a rule graph for the target game using the method described by Genesereth [29] and summarized in Section 3.1.1. We then find a mapping from the symbols of the target game to those of a source game by using the LMap algorithm. We have modified LMap by completely ignoring mapping for symbols belonging to long numeric chains. In GDL, consecutive numbers must be defined one at a time by the application of a successor function (e.g. `(succ 1 2)`, `(succ 2 3)`, and so on). This creates a long chain of connected symbols that is easy to recognize without the need for costly mapping. If the successor function that defines a chain in the target game

has been mapped to the successor function for a chain in the source game, we can then proceed up each chain, mapping the elements of one to the other.

Obtaining a symbol mapping is necessary for the initialization of the symbol counting and history heuristics, since each of these assign value to particular combinations of symbols encountered in the game. For example, we may have assigned a high weight to counting the number of black pieces in the source game, but after obfuscation, this may correspond to counting the number of occurrences of 'firetruck' in the target game. Without a way to map from 'bp' to 'firetruck', we would not be able to transfer heuristic values, even if the target game was actually the same as the source before obfuscation. After performing the mapping and loading heuristic values from file, the TI-GMCTS agent is initialized, and may use the rest of the start clock to perform Guided MCTS.

In this chapter, we assume that an appropriate source game is known in advance for the given target game. For use in competition, a TI-GMCTS agent would need some method for selecting a source game from its inventory of previously played games, which is a problem addressed in the next chapter.

4.2.5 Data Storage

In this section, we detail the specifics of storing rule graphs, raw heuristic data, and final linear regression values to disk.

Rule Graphs: Most of the information saved for a rule graph is an adjacency list specifying the edges of the graph. This is done by assigning a unique ID to each node in the graph. Although the IDs count upward from 0, the ordering of nodes is not inherently important. It is simply the order that the nodes were originally generated.

The structure of the file is as follows:

- Line 1: Symbol IDs corresponding to the players' roles.
- Line 2: IDs for symbols that are part of a numeric chain.
- Line 3: Number chain IDs in order, where the first element is the ID of the successor function, the second element is the ID of the first symbol in the chain, and so on. Different chains are separated by '*'.

- Lines 4-end: One line for each node in the rule graph. Each line contains the ID of the node, a number representing its colour, a String containing its name, the number of arguments it accepts (or -1 if it does not accept arguments), and then the IDs of each of its child nodes.

Figure 4.2 shows the topmost part of the file that contains the rule graph for Base Checkers.

Raw Heuristic Data: For each game played, heuristic data is saved in the form that is needed for simulation initialization. For most heuristics, that means saving the data from each rollout that was conducted over the course of play for use as a single data point in linear regression. (The history heuristics are the exception, for which only averages across all rollouts are saved.)

The structure of the resulting file is as follows:

- Line 1: The number of players
- Line 2: Maximum and minimum numbers of occurrences for each *(symbol, fact, position)* triple. Data is given in the order: symbol ID, parent ID, position, maximum, minimum. Different triples are separated by '*'.
- Line 3: Symbol count data for every rollout made in the game. Different rollouts are separated by '*'. For each rollout, the first two values are the reward received by each player. Then data is given for every triple in the form: symbol, parent fact, position, total number of occurrences across all states in the rollout, number of states in the rollout. Different triples are separated by '#'.
- Line 4: Mobility data for every rollout made. Different rollouts are separated by '*'. Data consists of three values pertaining to each player, one after another. These values are the reward received, mobility score totalled across all states in the rollout, then the number of states in the rollout.
- Lines 5-6: One line per player assigning an ID to each unique move played. These are used reduce the size of later lines. For each move, the ID is listed, followed by the symbol IDs that make up the move in parentheses.
- Lines 7-8: One line per player giving general history heuristic data. Data for separate moves is separated by '*', and is of the form: move ID, total reward, number of wins, number of losses, number of draws, total number of occurrences.

```
 1: 2 5
 2: 3585 3075 3589 3079 3593 3083 12 3597 3087 3601 3091 3605 22 3095 3609 ...
 3: 939 2356 12 46 22 28 34 62 40 68 2730 2737 2733 363 2880 2884 2888 2892 ...
 4: 0 5 role -1 1
 5: 1 16 red -1
 6: 2 22 red -1 1 1281 1026 2114 1987 2371 1156 1349 1161 1225 2443 1294 ...
 7: 3 5 role -1 4
 8: 4 16 black -1
 9: 5 22 black -1 1344 2563 4 2438 967 2567 1992 2376 1995 1166 2448 916 ...
10: 6 8 init -1 7
11: 7 16 cell -1 8 11 14
12: 8 16 a -1
13: 9 22 a -1 32 289 20 228 2612 38 8 26 330 2570
14: 10 18 cell 1 259 1796 1541 8 264 269 782 274 1811 20 279 1047 26 284 ...
15: 11 16 1 -1
16: 12 24 1 -1 963 2756 582 265 2825 74 11 2827 975 2646 1049 539 2843 2845 ...
17: 13 18 cell 2 260 1797 1542 265 11 270 784 275 1812 21 280 1049 27 285 ...
18: 14 16 b -1
19: 15 22 b -1 1922 132 1543 137 2571 269 1933 2573 14 142 786 147 403 1813 ...
20: 16 18 cell 3 256 1024 261 1798 1543 266 14 271 786 276 1813 23 281 1050 ...
21: 17 23 cell -1 258 1795 1540 7 263 10 268 13 781 16 273 1810 19 278 1046 ...
22: 18 8 init -1 19
23: 19 16 cell -1 20 21 23
24: 20 16 a -1
25: 21 16 3 -1
26: 22 24 3 -1 2849 131 2595 2819 2851 2788 2597 2821 2758 270 79 275 21 183 ...
27: 23 16 b -1
28: 24 8 init -1 25
29: 25 16 cell -1 26 27 29
30: 26 16 a -1
31: 27 16 4 -1
32: 28 24 4 -1 2656 2816 2785 2818 2755 2852 2598 2854 136 2600 105 209 51 84 ...
33: 29 16 b -1
34: 30 8 init -1 31
```

Figure 4.2: The first 34 lines of the (non-obfuscated) rule graph for Base Checkers, saved to disk. Ellipses indicate that a line continues, but was truncated for space.

```
 1: 2
 2: 15 17 3 40 8 * 12 365 2 2 1 * 46 365 2 2 1 * 2733 365 2 2 1 * ...
 3: 0 100 15 17 3 1807 102 # 46 365 2 8 8 # 2733 365 2 26 22 # ...
 4: 0 -50.0 50 100 148.0 50 * 0 104.0 50 100 0.0 50 * 0 88.0 50 ...
 5: 0 ( 395 231 73 62 15 40 ) 1 ( 395 947 203 34 177 62 ) 2 ( 395 ...
 6: 0 ( 427 965 99 22 15 34 99 40 ) 1 ( 427 965 125 28 73 46 9 28 ) ...
 7: 465 17250 162 30 21 213 * 1989 1276250 11748 1428 2029 15205 * ...
 8: 465 13850 114 147 49 310 * 1989 244250 1428 11748 2029 15205 * ...
 9: 0 157800 1457 253 242 1952 * 1 89850 822 136 153 1111 * 2 119750 ...
10: 0 700 5 16 4 25 * 1 300 1 4 4 9 * 2 50 0 1 1 2 * 3 2150 19 2 5 ...
11: 7 7 2 5
12: 0 100 947 48 516.0 193.5 216.0 409.5 0 0 0 0 * 292 102 844.0382 ...
```

Figure 4.3: All necessary lines of raw heuristic data collected from a single game of Base Checkers, saved to disk. Ellipses indicate that a line continues, but was truncated for space.

- Lines 9-10: One line per player giving specific history heuristic data. Data is of the same form as for the general history heuristic, except that the move IDs refer to those defined on lines 5-6.
- Line 11: Board information. In order, the values are the length in the X dimension, the length in the Y dimension, the minimum length of lines for which data was recorded, and the maximum length of lines for which data was recorded.
- Line 12: Board data for each symbol appearing on the board in each rollout. Rollouts are separated by '#', and within them, data for each symbol is separated by '*'. At the beginning of each rollout, the reward received by each player is recorded. After that, data is of the form: symbol ID, number of states that the symbol appears, average centre distance, average X-side distance, average Y-side distance, average corner distance, and then the average number of lines of each length between the limits specified on Line 11.

Figure 4.3 gives this file for one game of Base Checkers.

Linear Regression Values: Using the raw heuristic data files, we pre-compute one file of heuristic values. This is done in the same way as in simulation initialization (i.e. mostly via linear regression), by treating the rollouts across all raw data files as if they came from

```
 1: 2 2 5
 2: 0.0045704463482828285 15 17 3 1.6529764520938957 ...
 3: -0.004570446348282809 15 17 3 -1.652976452093895 ...
 4: 13.827304758835215 38.51210814119355 855895 0.6895509808559308
 5: 13.88670588082707 36.20608913241265 855895 0.6892079200568458
 6: 465 2807400 26528 6020 3092 35640 * 1989 42186400 363357 ...
 7: 465 5185300 42673 16755 18360 77788 * 1989 43403100 375524 ...
 8: 6 395 231 73 62 15 40 10140300 89670 85523 23466 198659 * 6 ...
 9: 8 427 965 99 22 15 34 99 40 667700 6453 527 448 7428 * 8 427 ...
10: 947 0.07442320737710584 -9.000040990356409 854449.0 ...
11: 947 -0.07442320737710588 109.00004099035644 854449.0 ...
12: 947 0.06574846012120263 30.94656356362593 854449.0 ...
13: 947 -0.06574846012120267 69.05343643637408 854449.0 ...
14: 947 -0.056093772235874015 61.91209395319058 854449.0 ...
15: 947 0.056093772235873994 38.08790604680943 854449.0 ...
16: 947 0.019781667674842032 39.37526926069999 854449.0 ...
17: 947 -0.019781667674842025 60.624730739300006 854449.0 ...
18: 947 1.145002153149548 36.836893472840565 854449.0 ...
19: 947 -1.1450021531495485 63.163106527159435 854449.0 ...
20: 947 4.839487608901394 47.8267373560199 854449.0 ...
21: 947 -4.839487608901394 52.1732626439801 854449.0 ...
22: 947 11.521198616018491 49.3193130052279 854449.0 ...
23: 947 -11.521198616018497 50.6806869947721 854449.0 ...
24: 947 20.394402369157586 49.34266271807081 854449.0 ...
25: 947 -20.39440236915759 50.65733728192919 854449.0 ...
```

Figure 4.4: All necessary lines of linear regression values, saved to disk. Ellipses indicate that a line continues, but was truncated for space.

the same game. The resulting heuristic values are then written to one final file, which can be loaded by a TI-GMCTS agent during initialization.

The form of this file is as follows:

- Line 1: The number of players, the minimum length for lines of pieces, then the maximum length for lines of pieces.
- Lines 2-3: One line per player giving symbol counting heuristic values. The first value on a line is the average Pearson correlation coefficient (r-value) across all *(symbol, fact, position)* triples. After that, data is given for each triple separated by '*', in the form: symbol ID, parent ID, position, slope, intercept, number of data points, r-value.
- Lines 4-5: One line per player for mobility heuristic values, in the form: slope, intercept, number of data points, r-value.
- Lines 6-7: One line per player for general history heuristic values. Different moves are separated by '*', and values are given as: move ID, total reward, number of wins, number of losses, number of draws, total number of data points.
- Lines 8-9: One line per player for specific history heuristic values. Different moves are separated by '*'. The first value gives the arity of the move tuple, n, and the next n values are the symbols making up the tuple. After that, values are given as: total reward, number of wins, number of losses, number of draws, total number of data points.
- Lines 10-11: One line per player for the centre distance board heuristic. Values for each symbol are separated by '*', and given as: symbol ID, slope, intercept, number of data points, r-value.
- Lines 12-end: One line per player for each of the X-side distance, Y-side distance, corner distance, and each possible length for lines of pieces. All of these are specified in the same form as the centre distance.

Figure 4.4 shows the heuristic data file that is loaded by TI-GMCTS when it plays Base Checkers, or any of its variants.

4.2.6 A Note on Direct State Comparison

Rather than using an assortment of general heuristics, an earlier version of this work attempted to compare states and actions directly to those seen in a previous game. For actions, this effectively amounted to use of the specific and general history heuristics, but state information was utilized very differently from the methods we have described in this chapter. For each source game, we kept data for the 10,000 nodes most frequently visited by MCTS. During search in the target game, a most similar source state was identified as the one with the largest number of shared facts, and its average reward was used as a measure of goodness. (In games with a board, it is typical to have one fact for the state of each playable square.)

It was hoped that even though 10,000 is many orders of magnitude smaller than the size of the state space for games of interest (Checkers, with 32 squares each containing one of five possible symbols, has an upper bound of $2.3 * 10^{22}$ possible states), selecting those most visited would provide useful guidance, especially in the early parts of a game where the number of reachable states is much smaller.

While we did find that this transfer agent was able to outperform UCT with statistical significance on Checkers and some of its variants, we found the contribution from direct state comparison to be negligible. In the first few moves, where matches were good, the transferred reward information was not very informative due to the large number of moves left to be played. Afterwards, the quality of state matches degraded too rapidly for them to be useful.

Intuitively, part of the reason for this is a misalignment between what was considered to be a similar state, and what actually makes a state good or bad. For example, take any state and swap one of the red pieces for a black piece. Humans would identify this as a two-piece swing, a relatively important change, while our direct state comparison could only see that 1/32 facts representing squares had changed. On the other hand, each piece translated by one space, would be seen as a difference in 2/32 facts (since one square's occupancy would change from a piece to empty, and another's would change from empty to a piece).

Recognizing this misalignment, we abandoned direct state comparison in favour of the general heuristics described in Section 4.2.1, which capture more focused characteristics of a game state that are more likely to be relevant in state evaluation.

4.3 Experimental Evaluation

The following experiments compare the performance of TI-GMCTS against SI-GMCTS and a UCT baseline in variants of Checkers and Breakthrough. To avoid introducing bias, these agents share code for interacting with GDL and implementing MCTS. TI- and SI-GMCTS agents also share code for the implementation of heuristics. All code was written in Java, and is built off of the GGP Base Package [64]. Each agent ran as a single-threaded process on a Ubuntu 22.04 machine with a 12th Gen Intel i7-12700 processor.

For a competitive GGP agent using TI-GMCTS, it would be desirable to swap to SI-GMCTS (or even UCT) if negative transfer were discovered during initialization (by comparing the expected reward to the actual reward seen in rollouts, for example). For these experiments, however, we are interested in testing TI-GMCTS on its own merit, and not a larger system that incorporates it. Our TI-GMCTS agent therefore proceeds as normal when negative transfer occurs, and we have included the Reversed Checkers variant specifically to highlight this case.

We will refer to (X-Y) time controls, where X indicates the start clock, and Y indicates the time per move afterward. For our main experiments, we have used (10-10) or (5-5) time controls, which are very short, even by the standards of GGP. This is because we want to examine the regime in which the initialization of an SI-GMCTS begins to experience enough variance to impact its performance. Although time controls this short would probably not have been run in past GGP competitions, we believe that it is within the spirit of GGP research to push for ever faster agents, particularly as computers become more powerful over time.

4.3.1 Checkers Variants

For each experiment involving Checkers, we have used the base version of checkers available in the GGP repository [1] as our source game, and when variants of Checkers are referenced, it is this base version from which they are modified. Although checkers is far from the most difficult game that AI researchers have developed agents to play, this version makes for an interesting test bed for a few reasons. First, jumping is not mandatory, as it is in the version of the game most often played by humans. This keeps the branching factor large, and avoids bogging down the state machine with additional rules. Second, a reward is not assigned until the end of the game, and that reward is all-or-nothing (except in the case of a draw, where reward equal to half of a win is given). This makes the problem difficult through a very sparse reward signal. A game ends when one player has no pieces

remaining, when the player to move cannot do so, or when 102 turns have passed. In all cases, the winner is the player with the most pieces remaining. If pieces are equal, it is a draw.

To produce games that are similar to Checkers, but have altered state/action spaces, we have used game modifications described in [45] (and summarized in Section 4.1, wherein each of these variants would have required its own pre-defined detector. They are as follows:

- **Base**: The unmodified base game.
- **Torus**: The board topology is that of a torus (i.e. pieces can move from the left edge of the board wraps around to the right edge, and the top edge wraps to the bottom).
- **Big Board**: The size of the board has been increased to 10X10. Pieces have been added to fill 3 rows for each player.
- **Small Board**: The size of the board has been decreased from 8X8 to 6X6. The number of pieces for each player has been reduced to fit in 3 rows on the smaller board.
- **No Middle**: The middle row of pieces has been removed for each player.
- **Reversed**: Reward values have been swapped. A win in standard checkers is now a loss, and vice versa.

TI-GMCTS and SI-GMCTS agents were run with a heuristic decay of $\alpha = 0.9$, a reward discount factor of 0.999, and an exploration parameter of $c = 0.4$.

4.3.2 Breakthrough Variants

Breakthrough is a board game based loosely on the movement of pawns in Chess. Each turn, a player moves one pawn one square, either directly forward or forward diagonally. An opponent's piece may be captured if and only if a diagonal move is made, as in Chess. Unlike Chess, a capture is not required to move diagonally. The game is won by the first player to move one of their pawns all the way across the board. Draws are not possible. Compared to Checkers, Breakthrough features a larger average branching factor, but allows rollouts to terminate much more quickly.

We test the following variants of Breakthrough:

	TI-GMCTS vs. UCT (5-5) time control		
Game	Wins	Losses	p-value
Base	79	17	$< 0.00001^{***}$
Torus	91	3	$< 0.00001^{***}$
Big Board	91	3	$< 0.00001^{***}$
Small Board	66	27	0.00003^{***}
No Middle	78	16	$< 0.00001^{***}$
Reversed	5	89	1.0

	TI- vs. SI-GMCTS (10-10) time control			TI- vs. SI-GMCTS (5-5) time control		
Game	Wins	Losses	p-value	Wins	Losses	p-value
Base	46	44	0.46	60	28	0.0004^{***}
Torus	61	34	0.004^{***}	68	27	0.00002^{***}
Big Board	53	42	0.15	54	40	0.09^{*}
Small Board	46	39	0.26	51	31	0.02^{**}
No Middle	43	48	0.89	63	31	0.0006^{***}
Reversed	11	76	1.0	4	88	1.0

Table 4.1: Results for TI-GMCTS vs. UCT and SI-GMCTS on variants of Checkers, with Base Checkers as the source game, for 100 trials each.

- **Small**: Each player has 2 rows of pawns on a 6X6 board.
- **Standard**: Each player has 2 rows of pawns on an 8X8 board.
- **Holes**: The game is Standard, but some squares are "holes" that pawns cannot enter.
- **1.5 Line**: The game is Standard, but half of the pawns in the forward-most row have been removed for each player.
- **2.5 Line**: The game is Standard, but an extra half row of pawns have been added for each player.

For experiments involving variants of Breakthrough, we have used Small Breakthrough as the source game. This simulates cases where our agent has not seen the standard version of a game, and must instead transfer directly between variants. TI-GMCTS and SI-GMCTS agents were run with a heuristic decay of $\alpha = 0.99$, a reward discount factor of 0.999, and an exploration parameter of $c = 0.4$.

Game	TI-GMCTS vs. UCT (5-5) time control		
	Wins	Losses	p-value
Small	67	33	0.0004***
Standard	83	17	< 0.00001***
Holes	73	27	< 0.00001***
1.5 Line	73	27	< 0.00001***
2.5 Line	77	23	< 0.00001***

Game	TI- vs. SI-GMCTS (10-10) time control			TI- vs. SI-GMCTS (5-5) time control		
	Wins	Losses	p-value	Wins	Losses	p-value
Small	56	44	0.14	67	33	0.0004***
Standard	87	13	< 0.00001***	83	17	< 0.00001***
Holes	76	24	< 0.00001***	73	27	< 0.00001***
1.5 Line	72	28	< 0.00001***	82	18	< 0.00001***
2.5 Line	82	18	< 0.00001***	75	25	< 0.00001***

Table 4.2: Results for TI-GMCTS vs. UCT and SI-GMCTS on variants of Breakthrough, with Small Breakthrough as the source game, for 100 trials each.

4.3.3 Results

Before comparing TI-GMCTS to SI-GMCTS, we first establish that our set of general heuristics and the resulting combined evaluation function are effective versus a UCT baseline. TI-GMCTS was played against UCT on (5-5) time controls for 100 iterations on each of the six Checkers variants and five Breakthrough variants. Results are given by the top portion of Tables 4.1 and 4.2. 'Wins' refers to wins for the TI-GMCTS agent, and 'p-value' is the result of a binomial test whose null hypothesis is that the likelihood of the TI-GMCTS winning is 50%. In tables, we mark p-values with asterisks to denote the level of statistical significance. If $p < 0.1$, we use one asterisk ('*'), if $p < 0.05$, we use two ('**'), and if $p < 0.01$, we use three ('***'). Draws were omitted from this test. We found that TI-GMCTS performs consistently better than UCT across all games, except for Checkers Reversed.

We ran TI-GMCTS against SI-GMCTS for a series of 100 games on each of the six Checkers variants and five Breakthrough variants, under both (10-10) and (5-5) time controls. Results of these runs are given by the bottom portion of Tables 4.1 and 4.2, respectively. For Checkers, we observe that TI-GMCTS only outperforms SI-GMCTS in one Checkers variant with statistical significance at (10-10) time controls, but does so in five

Heuristic	Archive value	Base avg	Base std	Torus avg	Torus std	Big avg	Big std
Mobility (10-10)	0.69	0.63	0.08	0.64	0.09	0.64	0.11
Mobility (5-5)	0.69	0.64	0.13	0.62	0.15	0.63	0.15
Red Pcs. (10-10)	0.25	0.22	0.13	0.0	0.17	0.45	0.17
Red Pcs. (5-5)	0.25	0.19	0.21	-0.01	0.28	0.43	0.21
Red Kings (10-10)	0.39	0.35	0.15	0.32	0.15	0.29	0.22
Red Kings (5-5)	0.39	0.37	0.18	0.29	0.26	0.32	0.23
Dbl. Jump (10-10)	62.0	56.87	9.44	51.89	9.55	57.05	11.6
Dbl. Jump (5-5)	62.0	54.38	12.48	52.5	12.79	56.84	15.29

Heuristic	Small avg	Small std	No Mid avg	No Mid std	Reversed avg	Reversed std
Mobility (10-10)	0.67	0.07	0.69	0.08	-0.66	0.07
Mobility (5-5)	0.66	0.11	0.67	0.14	-0.63	0.12
Red Pcs. (10-10)	-0.05	0.13	0.02	0.14	-0.23	0.15
Red Pcs. (5-5)	-0.06	0.16	0.0	0.22	-0.23	0.22
Red Kings (10-10)	0.44	0.1	0.45	0.11	-0.38	0.12
Red Kings (5-5)	0.44	0.17	0.43	0.18	-0.39	0.19
Dbl. Jump (10-10)	58.83	7.75	62.12	9.13	41.9	8.29
Dbl. Jump (5-5)	61.1	10.93	62.12	15.59	43.95	13.36

Table 4.3: Selected heuristic values for SI-GMCTS from the perspective of the red player. For the mobility and symbol counting (red pieces, red kings) heuristics, r-values are reported. For double jumps, the average reward is reported. Archive value refers to the values loaded by TI-GMCTS.

Checkers variants at (5-5) time controls with a significance of at least $p < 0.1$. As expected, TI-GMCTS significantly underperforms relative to SI-GMCTS on Reversed Checkers at all time controls. For Breakthrough, we find that TI-GMCTS outperforms SI-GMCTS with statistical significance in four variants at (10-10) time controls, and in all five variants at (5-5) time controls. These results are discussed further in Section 4.4.

4.4 Discussion

From the results of testing TI-GMCTS against SI-GMCTS, we observe that TI-GMCTS generally has an advantage in short time controls and games with a large branching factor. The former is most evident in Checkers (and also in Small Breakthrough), where moving

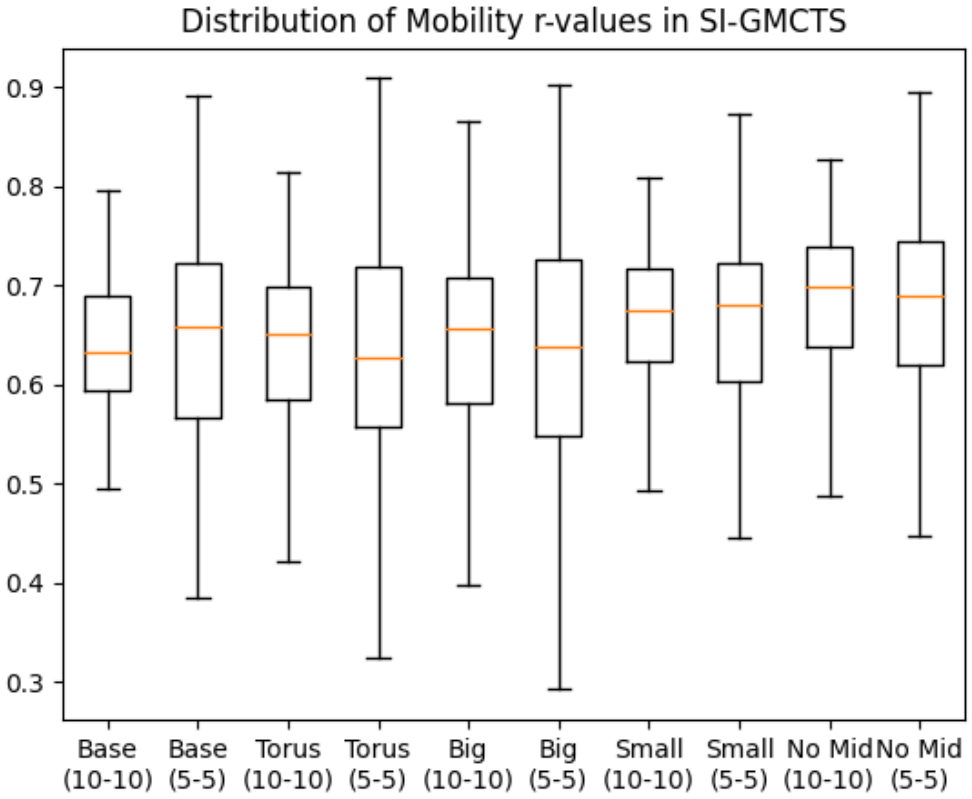

Figure 4.5: Distribution of the mobility heuristic r-value for SI-GMCTS

from (10-10) to (5-5) time controls creates a statistically significant difference in four of the five variants where there previously was not. The latter can be observed by comparing performance in the source game (Base Checkers, Small Breakthrough) to those with a larger branching factor (Torus Checkers, Big Checkers, all Breakthrough variants). In all of these cases, TI-GMCTS outperforms SI-GMCTS at both time controls, and this difference is statistically significant for every game, though the result for Big Checkers was less significant at $p = 0.09$. Suspecting that this exception was an outlier, we ran 100 more games of Big Checkers on (5-5) time controls with a result of 63W/30L, which gives a combined record of 117W/70L, and a p-value of 0.0001. This second result should not supplant the original; it merely helps to explain the exception to our observations in this discussion. In contrast, TI-GMCTS does not get as large an advantage in games with a smaller branching factor than the source game (Small Checkers, No Middle Checkers).

We can explain these observations by noting that TI-GMCTS loads its heuristic parameters from file, which ensures that they are the same every time. SI-GMCTS, on the other hand, relies on its initial simulations to be representative of the game it is playing. When the start clock is short, and the number of those simulations is diminished, more luck is required to achieve good heuristic parameter values. Similarly, games with a larger branching factor do not allow rollouts to cover as much of the search space. To support this hypothesis, we tracked the standard deviation for a selection of important heuristic values in Checkers variants played by SI-GMCTS, including mobility, piece counts, and the history heuristic for double-jumping.

Table 4.3 gives the standard deviation for a selection of important heuristic values, and Figure 4.5 shows the distribution of the mobility heuristic r-value for five of the games. (Reversed Checkers follows the same pattern, but its values are negative.) We observe from these data that the standard deviation of each of these values increases consistently, and often substantially, when time controls are decreased from (10-10) to (5-5). This is consistent with the hypothesis that heuristic parameters become increasingly unreliable for SI-GMCTS when time is short.

We have so far largely ignored the poor performance of TI-GMCTS on Reversed Checkers. As noted in Section 4.3.1, this variant was included to highlight the worst case scenario for TI-GMCTS, in which its heuristics actively guide it toward bad moves. It was therefore expected that TI-GMCTS would underperform compared to SI-GMCTS and UCT, and the extent of that underperformance serves to showcase that this set of general heuristics can sabotage an agent when they are not attuned to the game being played. A competitive GGP agent that utilizes TI-GMCTS needs to have a back-up system to avoid this outcome. For example, cases of extreme heuristic misalignment can be identified during the start clock, and SI-GMCTS can be used instead. This solution is explored in greater detail in Section 5.3.

We ran a few experiments attempting transfer from Checkers to Breakthrough, and vice versa, for TI- vs. SI-GMCTS using (5-5) time controls. From Table 5.1, we observe that the distance between the GDL descriptions of these games is quite high, which resulted in a poor quality mapping. There is, for instance, no counterpart in Breakthrough for a double-jump in Checkers, which is an important heuristic feature. Transferring from Breakthrough to Checkers produced a record of 51W/41L (p-value $= 0.17$), while the reverse direction resulted in a record of 79W/21L (p-value < 0.00001). We speculate that it may be more advantageous to transfer from a game with complex rules to one with simpler rules than the reverse, but more investigation is required to support this hypothesis.

4.5 Conclusion

We have presented a method for transferring heuristic knowledge between similar, but distinct games within the GGP framework. For any set of general heuristics that would normally require costly initialization during a game's start clock, this method allows heuristic parameters to instead be loaded from a pre-computed archive. Through experimentation on variants of Checkers and Breakthrough, we have shown that the resulting heuristics are robust to small changes in the state/action space, and that transfer provides the most benefit when short time controls or a large branching factor make initial simulations unreliable.

Chapter 5

Greater Generality

The previous two chapters of this thesis developed the pieces necessary to show that graph-based mapping between two similar games is possible on the short time scales of GGP, and that it can be used to profitably transfer knowledge from one to the other. To do so, we worked within a manageable scope with regard to games played, heuristics used, and assumptions made. In this chapter, we will push at the edges of that scope, showing that new methods can be effectively slotted into the old, and revisiting previously deferred problems.

To that end, this chapter will examine four topics. First, we revisit the problem of source game selection. By its nature, solutions are troublesome to evaluate because it is a problem that is as large as one makes it, as will be discussed in Section 5.1. However, to prove that the methods we have discussed in the preceding chapters may realistically be used in a GGP setting, it is necessary to show that there exists at least one fast solution to this problem.

Second, we integrate a new group of heuristics into our methods to demonstrate the modularity of our approach. In general, this work is not about advocating for any particular set of general heuristics. Rather, we show that our methods may accommodate whatever heuristics are desired.

To observe the effect of the new heuristics, we first return our attention to variants of Checkers and Breakthrough, and then introduce variants of Connect Four as a new test bed. Connect Four is far from the most difficult game in the GGP base repository with respect to its branching factor and the size of its state space, but this is part of what makes it an interesting game to test. A game that is easy, is easy for both players, and it is difficult for one agent to distinguish itself from another in that setting. To show that

TI-GMCTS is generally effective on a variety of games, it behooves us to examine not just those games that are large enough to cause UCT to struggle, but also to show that we can improve on its performance even in games that it already plays well. Besides this, Connect Four is a worthy subject for other reasons, to be discussed in Section 5.2.2.

As our third topic, we return to another previously deferred problem: that of negative transfer. In particular, we have seen that an easy way to confound TI-GMCTS into playing (much) worse than UCT is to simply swap the reward values for a win and a loss, but otherwise leave the game unchanged. In this worst case scenario for TI-GMCTS, its transferred heuristics become actively detrimental, prioritizing the exploration of bad states. Although it would be possible to detect and counteract this one trick directly, doing so would not address any other possible cases of negative transfer. Instead, we demonstrate a method for abandoning transfer when it is sufficiently negative, whatever the cause, which is a necessity in expanding the generality of our previous work.

Finally, our fourth topic is an examination of a class of game that we have previously ignored: single-player games, or puzzles. In the last section of this chapter, we discuss their particular foibles, as well as our motivation for studying them. We introduce a new path-finding puzzle based on Checkers and show that transfer can be effective from a two-player source game to one that is single-player, as long as similar notions of a 'good' move may be applied.

The subject of this chapter is achieving greater generality, and that is a goal that can never be considered complete. We have addressed what we consider to be the most prominent outstanding shortcomings and unanswered questions of the preceding work. Further improvements and new ideas are deferred to future work in Section 6.1.

5.1 Source Game Selection

In previous chapters, we have assumed that an appropriate source game is known in advance for the given target game. To be a full GGP agent, capable of performing tasks on demand or taking part in competition, a TI-GMCTS agent needs some method for selecting a source game from its inventory of previously played games. However, as was alluded in the introduction to this chapter, the nature and difficulty of this problem are not easily defined. For a given target game, selecting the closest game from the library of those that it has previously encountered is easiest and least expensive when the library is small. On the other hand, a larger library means a greater likelihood of finding a better source game from which to transfer, leading to higher performance. There are trade-offs that must be

Target Game	Distance to						Time (ms)
	Checkers	Break-through	8 Queens LG	Chess	Tic-Tac-Toe	Connect Four	
Checkers, Torus	**0.003**	0.663	0.737	0.517	0.672	0.632	38
Checkers, Big	**0.052**	0.676	0.747	0.501	0.682	0.642	38
Checkers, Small	**0.050**	0.646	0.733	0.539	0.656	0.625	40
Checkers, No Mid	**0.005**	0.660	0.742	0.518	0.671	0.636	38
Checkers, Reversed	**0.000**	0.662	0.742	0.517	0.671	0.631	38
Breakthrough, Small	0.710	**0.142**	0.590	0.801	0.383	0.495	40
Breakthrough w/ Holes	0.594	**0.196**	0.512	0.738	0.293	0.386	33
Breakthrough, 1.5 Rows	0.661	**0.028**	0.591	0.773	0.346	0.431	34
Breakthrough, 2.5 Rows	0.659	**0.034**	0.593	0.765	0.357	0.467	34
16 Queens, UG	0.772	0.641	**0.158**	0.845	0.539	0.600	37
31 Queens, LG	0.738	0.588	**0.129**	0.825	0.516	0.523	40

Table 5.1: Distances between game variants and various games in the GGP base repository, produced by LMap with depth = 1, averaged over 10 trials. LG (legal-guided) refers to a rules variant with a much smaller branching factor than UG (unguided).

made, both in what is reasonable and what is fair, and so we must leave a full treatment of how to build a library to future work.

We will, however, examine a method for selecting an appropriate source game, assuming that a library already exists. When LMap is run on the rule graphs of two games, it provides not just a mapping between their symbols, but also an estimated distance score representing the similarity of the games overall. For finding a useful mapping, we have run LMap at a search depth of five, which can take on the order of one second to run for two games of complexity similar to Checkers. Even for a small library, it would be prohibitively expensive to use LMap in this way to find the closest possible source game to a given target game. However, LMap's runtime decreases if we reduce the search depth, and the resulting reduction in accuracy may be acceptable since we need only a rough distance value to narrow our choices.

Table 5.1 gives the distances produced by using LMap at depth one to compare game variants to a selection of other games in the GGP base repository. In each row, the lowest distance (corresponding to the source game that would be selected) is in bold. Total time refers to the time taken to make all six necessary comparisons per row. From this table, we observe that each of the target game variants is correctly matched to the most appropriate source game in $\leq$ 40 ms, given this small catalogue of known games. This represents a negligible portion of the start clock, but will grow linearly with the number of games

catalogued.

For a larger library, where there is less likely to be one stand-out candidate, we recommend an iterated approach, in which LMap is run again for the best candidates at higher depth (and greater cost). Since experimental evaluation of this approach requires a canonical library, we also defer this to later work. For the remainder of this chapter, we will return to our assumption that an appropriate source game is known in advance, and the initialization time associated with LMap will be for a single game comparison at depth five.

5.2 Board Heuristics

Although we have previously avoided making assumptions about a game, like the presence of pieces and a board, such assumptions must be made in order to apply some of the general heuristics in the GGP literature. As a result, the heuristics that we will introduce and integrate into our existing work in this section will not be universally applicable to all games that may be described in GDL. They will, however, be applicable to both Checkers and Breakthrough, as well as Connect Four, which will be an important test bed in this section for reasons described in Section 5.2.2.

We assume the presence of a two-dimensional rectangular board with some number of rows and columns, whose squares can be occupied by any symbol. We do not assume any further common properties of board games, such as ownership of various symbols (pieces) by a player. From the board heuristics listed by Stephenson et al. [74], this narrows our selection to five:

- **Centre distance**: The average distance for symbols of one type to the centre of the board. Since there may be more than one type of symbol (e.g. Checkers uses separate symbols to represent red pieces, black pieces, red kings, and black kings), a value is separately assigned to each type. The following board heuristics are similarly applied to each different symbol.
- **Corner distance**: The average distance from a symbol to the closest corner of the board.
- **Vertical edge distance**: The average distance from a symbol to either the top or bottom of the board, whichever is closer.

- **Horizontal edge distance**: The average distance from a symbol to either the left or right side of the board, whichever is closer. (The axes representing vertical and horizontal may be chosen arbitrarily. It is only important that each represent one dimension of the board.)
- **Lines of like symbols**: On a two-dimensional board, symbols may be adjacent in any of the four cardinal directions, or the four diagonals. For this heuristic, we count the number of different lines of lengths two through five formed by symbols of the same type.

The first four of these heuristics are all variations on measuring distances between the locations of symbols and various features of the board. As a result, these heuristics are all handled very similarly, and will be treated as a group, going forward.

5.2.1 Method

Board Identification Before we can make use of any board heuristics, we must first identify which predicate describes the board (if indeed, there is a board), and the meanings of each of its arguments. Here we make the same assumptions as Banerjee et al. [8] and Kuhlmann and Stone [44]: that the board must be represented as a ternary predicate with two arguments representing rank and file, and the third, occupancy (although we make no assumptions on the ordering of these arguments). Further, the arguments corresponding to rank and file must always belong to a sequential set, like the number chains described in Section 2.2.2. These assumptions are crude, but effective for the games in which we are interested. (Once a board has been identified, we no longer care about how that identification was made, so a more sophisticated method may be dropped in, if desired.)

Unlike Banerjee and Kuhlmann, we would like to avoid the cost associated with searching the GDL code directly for such a predicate and determining the domain of its arguments. Instead, we observe the game states that appear over the natural course of MCT expansion and use them to update our notions of the board and, in particular, its length and width.

For some games, the whole board is immediately apparent from the initial state, which is the case for the versions of Checkers and Breakthrough found in the base GGP games repository. In Checkers, for example, every state contains a fact for every accessible square on the board, of the form `(cell c 2 bp)`, which, in this case, indicates that the square in the second rank of the C file contains a black piece. If the square were empty, `bp` would be

replaced by `b`, and there would still be a fact declaring the existence of square C2. Since we are able to identify that both `c` and `2` belong to number chains, and can see that eight adjacent elements of these chains are used in similar facts, we are able to determine the size of the board and how it is indexed from the initial state of the game.

However, making this determination is not always so easy, because we could instead have assumed a default state for each square. Squares that are in the default state (say, empty) need not be represented explicitly by facts, so the dimensions of the board may not be discoverable by checking the initial state alone. This is the case for the implementation of Connect Four in the GGP base repository, which is particularly troublesome because a game of Connect Four begins with no pieces in play. As a result, its initial state contains no information about the board.

To circumvent this issue, we look for a board at different points of a game. We examine one state that is a successor of the initial state and one terminal state produced by a rollout. If neither of these states contain any facts that match our criteria of a board, then we determine that there must not be a board, and make no further attempts to calculate board heuristics. If we do find a board, it is possible that its full length and width have not been revealed. In Connect Four, this occurs when no pieces occupy either the top row, or the left- or rightmost columns when the rollout terminates. We do, however, obtain minimum values for the dimensions of the board. As search progresses, we might discover a state that reveals the board to be larger than previously known. In that case, we record the new, larger dimensions.

Since board states are frequently encountered more than once during search, we keep caches for the heuristic values we calculate. For those heuristics that depend on the board dimensions (e.g. centre distance), we also note the known dimensions of the board at time each entry is made. Later, if we attempt to retrieve a value from the cache that was calculated with out of date board dimensions, we recalculate the heuristic value and update the cache.

There is some potential for a large number of cached values to be invalidated if the known board dimensions expand late into a search, but this does not seem to be likely, in practice. The boundaries of the Connect Four board were consistently discovered during the initialization period (and generally, within only a few rollouts), and even if empty squares were not explicitly represented in Checkers and Breakthrough, their initial states would give away their board dimensions due to piece placement.

Board Heuristic Calculation Like the symbol counting heuristic described in Section 4.2.1, the family of board heuristics all depend on the correlation of some property of a

type of symbol to the eventual reward received. So, these heuristics are each an ensemble of values, some of which are associated with each symbol that can appear on a board. At the end of a rollout, the board associated with each node on the path from root to terminal state is evaluated. For every symbol on the board, the Euclidean distance to the centre, nearest corner, nearest vertical edge, and nearest horizontal edge are calculated. These distances are then averaged across every matching symbol (e.g. all black kings).

Lines of pieces are counted such that one continuous line will not be counted more than once per possible length. For example, if we are looking for lines of length two and encounter a line of length four, we will still count it as one, even though it is composed of multiple unique lines of length two, in both the forward and backward directions. Practically, this means that we sweep over the board from the bottom left to the top right corner. Every time we encounter an unvisited symbol, we recursively search to the right, up, diagonally up and right, and diagonally up and left. When a line terminates, we increment the counters for all lengths less than or equal to its length, and mark all relevant board positions as visited.

Once heuristic values have been determined for every board in the playout, we average each across all boards, and associate those values with the final reward received. These form the (x,y) pairs for linear regression in the same manner as the symbol counting heuristic. By the time we need to make a heuristic evaluation of a board state, linear regression values will be known, either via simulation initialization or transfer. An overall heuristic value and weight may be calculated for each of the board distance heuristics and each line length identically to the process for the symbol counting heuristic described in Section 4.2.1.

Updated Selection Guidance In order to reduce noise in the heuristic evaluation function, we have made adjustments to the method for calculating it described in Section 4.2.1. During search, we do not calculate heuristic scores for single nodes in a vacuum. Rather, when a heuristic evaluation is made, it is for the purpose of comparing one action to each of the other possible actions. The states that result from these actions are siblings, each the child of some shared parent state. So, before passing a heuristic value to $UCB1_{Heur}$, we modify it in comparison with its siblings.

If some contribution to the heuristic score is the same across all siblings, we set the weight of that contribution to zero. For example, it may be the case that no possible action can change the player's mobility in a way that is different from any other action. In that case, the mobility heuristic is not providing useful information, so its contribution is zeroed. In this way, heuristics that are *never* meaningful (e.g. symbol counting in Connect Four) are permanently ignored.

After this step, we compute a single combined heuristic value, as before, and then perform one further action. We find the maximum heuristic value among all siblings and use it to normalize them so that the new maximum is equivalent to the maximum possible reward received (usually 100 points). This helps to better scale the heuristic portion of $UCB1_{Heur}$ to its other terms.

5.2.2 Evaluation

Connect Four In addition to those reasons previously discussed, Connect Four is an interesting game to test because the full set of previous heuristics were not able to produce performance significantly better than UCT. Upon inspection, we can immediately see that the symbol counting heuristic carries no useful information, since each player adds exactly one piece of their colour every turn, and this never changes. Likewise, the mobility heuristic is unlikely to be helpful because as columns fill up, they become unavailable to both players. The only potentially useful heuristic was the history heuristic for specific moves (in this case, corresponding to columns), since the central columns are generally more valuable, in a vacuum. However, this has not previously been enough information for either TI- or SI-GMCTS to distinguish themselves from UCT.

Intuitively, the board heuristics were expected to be more useful. In particular, forming a line of pieces is the goal of Connect Four, so the heuristics corresponding to lines of length two and three seemed promising. (Lines of length four and longer can only occur in terminal states, in which heuristics are not needed because any agent can see their true reward value.) This means that Connect Four presents an opportunity to study the effect of a mixed group of heuristics on performance, where some are useful, and others are simply noisy.

Our variants of Connect Four are as follows:

- **Base**: The standard version of Connect Four in the GGP base repository. There are eight columns into which pieces can be dropped, each of length six.
- **Tall**: As in Base Connect Four, there are eight columns, but each can hold twice as many pieces (twelve).
- **Large**: A larger board than Base Connect Four on both axes. There are twelve columns of length nine.

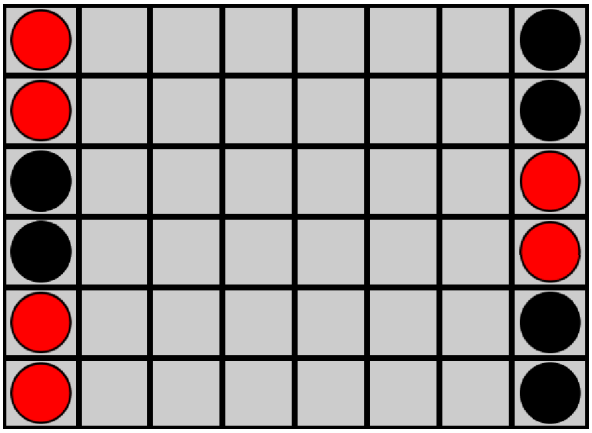

Figure 5.1: The initial state of Edge Connect Four.

- **Edge**: A standard Connect Four board, but each player begins with six pieces in play, occupying the outermost columns of the board. The pattern of these pieces is given by Figure 5.1.
- **36 Turn**: Base Connect Four, but the game is terminated after 36 turns have been played. If no player has made a line of four pieces, then the game is declared a draw, and both agents receive a reward of 50 points.

As for Checkers and Breakthrough, these variants of Connect Four are in line with modifications described by Kuhlmann and Stone [45], and each warps the game in a unique way. In Tall Connect Four, it takes much longer to fill up a column. This has a significant impact on search, since the branching factor diminishes more slowly than in Base Connect Four. Additionally, games (and by extension, rollouts) can be up to twice as long as the longest in Base Connect Four.

Large Connect Four imposes similar challenges, and additionally changes the action space. This increases the branching factor, and also misleads the history heuristic. In Base Connect Four, dropping a piece in the eighth column was not particularly desirable because of the limited number of lines in which a piece on the edge of the board can participate. In Large Connect Four, taking this action is considerably better because the eighth column is toward the centre of the board.

Edge Connect Four increases the value of playing near the edge of the board because there are already neighbouring pieces with which to start lines. This poses a challenge to the various distance-based board heuristics. In addition, the playable area is smaller,

which means that there is a reduced branching factor and smaller maximum game length, compared to Base Connect Four.

36 Turn Connect Four also has a decreased maximum game length, and in a way that has a significant impact on strategy. Frequently, a game of Base Connect Four is won when a player is forced to take a losing action (i.e. enable the opponent to immediately make a line of four) because there are no other actions left to take. In 36 Turn Connect Four, this is significantly less likely to occur because there are guaranteed to be at least three possible columns unfilled. Instead, games are most likely to be won by a player who exposes two threats at once, which cannot both be stopped in one turn.

Experimental Conditions The following experiments compare the performance of TI-GMCTS, SI-GMCTS, and UCT on Connect Four at particularly short time controls (5-5), (2-2), and (1-1)[1]. Since Connect Four is a simpler game than either Checkers or Breakthrough, this was the regime in which we see observed the most difference in agent performance. We also revisit Checkers and Breakthrough, using the same (5-5) and (10-10) time controls as previous sections, to see the effect of board heuristics on performance in those games. In all tests, both TI- and SI-GMCTS have access to all of their previous general heuristics, as well as the new board heuristics.

Each agent ran as a single-threaded process on a Ubuntu 22.04 machine with a 12th Gen Intel i7-12700 processor. For Checkers and Connect Four, heuristic decay for both TI- and SI-GMCTS agents was set to 0.9, and to 0.99 for Breakthrough. For every test condition, we ran 100 trials, except in the case of game variants that are especially prone to draws. For those (36 Turn Connect Four and Small Checkers), we ran 200 trials. (This will also be the case for the experiments of Section 5.3.)

5.2.3 Results

In this section and those that follow, we use the same conventions for tables as we have in previous chapters. 'Wins' refers to wins for the first agent listed, and 'p-value' is the result of a binomial test whose null hypothesis is that the likelihood of the first agent winning is 50%. In tables, we mark p-values with asterisks to denote the level of statistical significance. If $p < 0.1$, we use one asterisk ('*'), if $p < 0.05$, we use two ('**'), and if $p < 0.01$, we use three ('***'). Draws are omitted from significance tests.

[1] As in previous chapters, we use (X-Y) notation for time controls, where X indicates the start clock, and Y indicates the time per move afterward.

	SI-GMCTS vs. UCT (1-1) time control			SI-GMCTS vs. UCT (2-2) time control			SI-GMCTS vs. UCT (5-5) time control		
Game	Wins	Losses	p-value	Wins	Losses	p-value	Wins	Losses	p-value
Base	52	45	0.27	51	48	0.42	49	49	0.54
Tall	53	47	0.31	50	100	0.54	52	48	0.38
Large	41	59	0.972	30	100	1.0	49	51	0.62
Edge	44	54	0.87	45	50	0.73	51	46	0.34
36 Turn	57	61	0.68	37	65	0.998	44	47	0.66

	TI-GMCTS vs. UCT (1-1) time control			TI-GMCTS vs. UCT (2-2) time control			TI-GMCTS vs. UCT (5-5) time control		
Game	Wins	Losses	p-value	Wins	Losses	p-value	Wins	Losses	p-value
Base	60	38	0.017**	58	38	0.026**	48	50	0.62
Tall	67	32	0.0003***	60	40	0.028**	48	52	0.69
Large	49	51	0.62	45	55	0.86	42	58	0.96
Edge	58	39	0.034**	56	41	0.077*	49	46	0.42
36 Turn	102	39	< 0.00001***	74	43	0.0027***	64	40	0.012**

	TI- vs. SI-GMCTS (1-1) time control			TI- vs. SI-GMCTS (2-2) time control			TI- vs. SI-GMCTS (5-5) time control		
Game	Wins (W)	Losses (L)	p-value	W	L	p-value	W	L	p-value
Base	61	36	0.0072***	63	34	0.0021***	57	42	0.080*
Tall	67	33	0.0004***	69	31	0.00009***	55	45	0.18
Large	72	28	< 0.00001***	57	43	0.097*	50	50	0.54
Edge	56	43	0.114	47	48	0.58	51	49	0.46
36 Turn	73	64	0.25	83	40	0.00007***	60	31	0.0016***

Table 5.2: Results for games between TI-GMCTS, SI-GMCTS and UCT on variants of Connect Four, with Base Connect Four as the source game (for TI-GMCTS). Each condition was run for 100 trials, except for 36 Turn Connect Four, which was run for 200 trials.

Table 5.2 gives the results for TI-GMCTS, SI-GMCTS, and UCT playing against each other on the variants of Connect Four with all heuristics (including board heuristics) enabled. Looking first at the topmost table, which contains results for SI-GMCTS versus UCT, we draw the conclusion that SI-GMCTS generally performs no better than UCT at the time scales tested, and in a few cases, performs significantly worse. In particular, UCT wins Large Connect Four with $p < 0.05$ at both the (1-1) and (2-2) time controls, though not at (5-5). Given that Large Connect Four is the variant with the largest state space, this result is in keeping with our previous observation that SI-GMCTS suffers from high variance in its heuristics when it is short on initialization time. The only other statistically significant result in this table is another loss for SI-GMCTS at 36 Turn Connect four at a (2-2) time control. However, this may be a statistical anomaly, since neither the results for the (1-1) or (5-5) time controls are remarkable.

The middle portion of Table 5.2 gives results for TI-GMCTS against UCT. With a threshold for statistical significance of 0.05, we observe significant victories for TI-GMCTS in four out of five games at a (1-1) time control, three (nearly four) games at a (2-2) time control, and one game at a (5-5) time control. As we have previously seen, transfer confers the most benefit when time is too short for an agent to properly evaluate its possible moves via search. Conversely, there are no conditions under which UCT beats TI-GMCTS with $p < 0.05$, although Large Connect Four at (5-5) does breach the $p < 0.1$ threshold, with $p = 0.067$. In general, TI-GMCTS performs worse on Large Connect Four at every control than it does any other variant. Rather than variance in heuristic values (because there is no variance in values loaded from file), this may be attributable to a misalignment in them. For example, the history heuristic is actively misleading, as previously discussed.

From the bottom portion of Table 5.2, we find that TI-GMCTS has statistically significant ($p < 0.05$) victories over SI-GMCTS in three, three, and one game at (1-1), (2-2), and (5-5) time controls. SI-GMCTS answers with no significant victories of its own, even at a (5-5) control, where it is expected to be at its best, relative to TI-GMCTS. In fact, TI-GMCTS only has one losing record: a 47 to 48 loss in Edge Connect Four at a (2-2) time control.

Taking these results as a whole, we conclude that SI-GMCTS performs similarly to UCT on Connect Four and its variants, but if anything, is slightly worse. TI-GMCTS, on the other hand, clearly performs better overall than the other two, though this is not the case for every individual condition.

Table 5.3 gives results for TI-GMCTS versus UCT and SI-GMCTS on Checkers. These experiments are a repeat of those in Section 4.3, but with the addition of board heuristics. To see their effect, we may compare Table 5.3 to Table 4.1. Against UCT, we observe

	TI-GMCTS vs. UCT (5-5) time control			TI-GMCTS vs. UCT (10-10) time control		
Game	Wins	Losses	p-value	Wins	Losses	p-value
Base	76	15	$< 0.00001^{***}$	69	26	$< 0.00001^{***}$
Torus	94	1	$< 0.00001^{***}$	97	1	$< 0.00001^{***}$
Big Board	83	11	$< 0.00001^{***}$	78	18	$< 0.00001^{***}$
Small Board	130	33	$< 0.00001^{***}$	103	54	0.00006^{***}
No Middle	77	7	$< 0.00001^{***}$	68	23	$< 0.00001^{***}$

	TI-GMCTS vs. SI-GMCTS (5-5) time control			TI-GMCTS vs. SI-GMCTS (10-10) time control		
Game	Wins	Losses	p-value	Wins	Losses	p-value
Base	57	15	$< 0.00001^{***}$	47	25	0.0064^{***}
Torus	55	20	0.00003^{***}	58	22	0.00004^{***}
Big Board	51	28	0.0064^{***}	47	31	0.044^{**}
Small Board	79	35	0.00002^{***}	71	42	0.0041^{***}
No Middle	57	23	0.00009^{***}	47	35	0.11

Table 5.3: Results for TI-GMCTS vs. UCT and SI-GMCTS on variants of Checkers after the addition of board heuristics. For each condition, Base Checkers was used as the source game. For the Small Board condition, 200 trials were run. For every other condition, 100 trials were run.

	TI-GMCTS vs. UCT (5-5) time control			TI-GMCTS vs. UCT (10-10) time control		
Game	Wins	Losses	p-value	Wins	Losses	p-value
Small	77	23	$< 0.00001^{***}$	73	27	$< 0.00001^{***}$
Standard	92	8	$< 0.00001^{***}$	85	15	$< 0.00001^{***}$
Holes	79	21	$< 0.00001^{***}$	65	35	0.0018^{***}
1.5 Line	67	33	0.0004^{***}	63	37	0.0060^{***}
2.5 Line	100	0	$< 0.00001^{***}$	98	2	$< 0.00001^{***}$

	TI-GMCTS vs. SI-GMCTS (5-5) time control			TI-GMCTS vs. SI-GMCTS (10-10) time control		
Game	Wins	Losses	p-value	Wins	Losses	p-value
Small	76	24	$< 0.00001^{***}$	70	30	0.00004^{***}
Standard	83	17	$< 0.00001^{***}$	78	22	$< 0.00001^{***}$
Holes	87	13	$< 0.00001^{***}$	82	18	$< 0.00001^{***}$
1.5 Line	75	25	$< 0.00001^{***}$	69	31	0.00009^{***}
2.5 Line	89	11	$< 0.00001^{***}$	93	7	$< 0.00001^{***}$

Table 5.4: Results for TI-GMCTS vs. UCT and SI-GMCTS on variants of Breakthrough after the addition of board heuristics. Small Breakthroughwas used as the source game, and 100 trials were run for each condition.

that TI-GMCTS remains dominant, with records that are extremely significant in every Checkers variant, although this was already the case without board heuristics. Against SI-GMCTS, we do see some notable improvements. TI-GMCTS now has statistically significant victories over SI-GMCTS on the Big Board and Small Board variants at both (5-5) and (10-10) time controls, which was not previously the case. It also improved its (formerly losing) record on the No Middle variant at a (10-10) time control to one that is winning, though not quite to the level of statistical significance.

We can similarly compare the results of Table 5.4 to Table 4.2. Here, all conditions resulted in very statistically significant victories, with some variance in the absolute number of wins compared to Section 4.3.3. Of note, TI-GMCTS previously had a winning record against SI-GMCTS on Small Breakthrough, (10-10) condition, but not by margin of significance that it has in this more recent set of tests.

In summary, we find that the addition of board heuristics has enabled TI-GMCTS to outperform UCT overall on the family of Connect Four variants, and has done so without hampering performance on our previous test games. They too, see improvement.

5.3 Negative Transfer Protection

In Section 4.3.3, we found that reversing the goal values of a game so that winning states become losing, and vice-versa, was an extremely effective counter to TI-GMCTS. Were we to specifically detect this trick, it would be a simple matter to reinterpret our transferred linear regression values, and then continue as usual. However, the reversal of reward values is just the most extreme example of a larger problem, one with many possible causes that may not be so easily detected by examining the GDL code. Negative transfer can occur any time there is a problematic misalignment between our transferred heuristic values and those that would be naturally generated from the target game.

In this section, we describe and evaluate a method for detecting negative transfer, and subsequently bailing out of transfer altogether. When this occurs, TI-GMCTS is still at a disadvantage compared to an SI-GMCTS agent (for reasons to be discussed shortly), but is able to cut its losses and avoid the disastrously losing records of Section 4.3.3.

5.3.1 Method

For TI-GMCTS, negative transfer detection may be performed at the end of the initialization phase. Up to that point, initialization proceeds as usual: heuristic parameters are

initialized via transfer and the MCT is expanded for whatever time remains. During its search, the agent records heuristic and reward data in the same manner that an SI-GMCTS agent would. At the end of the start clock, this data is used to calculate a separate set of simulation-initialized (SI) heuristic parameters, as described for an SI-GMCTS agent in Section 4.2.1.

Unlike the transfer-initialized (TI) parameters, the SI parameters are guaranteed to correspond to real rewards seen in the current game, and are not negatively impacted by tricks like reward reversal. However, as seen in Section 4.3.3, they can also be noisy enough to degrade overall performance, which is a problem that worsens as time controls are shortened. With that in mind, we would prefer to use the TI values, as long as they appear to be reasonably applicable to the target game. We use the SI values to verify them.

For each individual general heuristic, there is an associated weight value that captures the predictive strength of that heuristic for eventual reward, as described in Sections 4.2.1 and 5.2.1. For most of the heuristics that we have used, this is a Pearson correlation coefficient (r-value). (The history heuristics are an exception, and we will revisit them later.) At the time that negative transfer detection is to occur, we use the symbol mapping method described in Section 4.2.1 to find pairs of matching TI and SI values, where needed. (Mobility does not require mapping, but all of the other heuristics we have described do.) These values of these pairs are expected to differ somewhat, even when the transferred value is a perfect fit. We will therefore only consider a TI value to be 'bad' when the following two conditions are met:

- The sign of the correlation is opposite that of the corresponding SI value, and
- The magnitude of the corresponding SI weight exceeds a minimum threshold (which is a tuning parameter).

On the other hand, we consider a TI value to be 'good' when:

- The sign of the correlation is the same as that of the corresponding SI value, and
- The magnitude of the TI weight exceeds a minimum threshold (which is a tuning parameter)[2].

[2]In principle, this threshold could be different from the previous one, but we have always used one value for both.

Heuristics that have weights of low magnitude are considered neither 'good' nor 'bad', but merely unimportant, as they will not contribute much to the heuristic evaluation function, even if their sign is incorrect.

For heuristics composed of smaller heuristics (e.g. the symbol counting heuristic has correlation coefficients associated with each different symbol), each of the lower level heuristics is deemed 'good', 'bad', or unimportant. A vote is then taken. If the number of 'bad' heuristics exceed the 'good', then the overarching parent heuristic is considered to be 'bad', and vice-versa for 'good'. Unimportant heuristics do not vote. (Although we have required a simple majority, the threshold for 'badness' need not be set at 50%.)

Once all high-level heuristics have been evaluated, each casts one final vote to determine the overall value of transfer. If 'bad' heuristics are in the majority, we determine that transfer is not worthwhile, and initiate a full swap of all TI heuristic parameters to the SI parameters that were found during initialization. Additionally, caches of heuristic values are purged, and the MCT is pruned back to its root node. From that point, the agent effectively becomes an SI-GMCTS agent that must start its search from scratch at the beginning of the first turn's clock. Additionally, it will have had less time to determine its SI values due to the initialization time required for transfer (running LMap, in particular). This clearly puts TI-GMCTS at a disadvantage compared to SI-GMCTS when a full swap occurs, but this is much preferred to working with actively misleading heuristics for the rest of the game.

If the final vote shows a 'good' majority, then a choice must be made. We can either elect to retain all TI values and proceed with TI-GMCTS as normal, or we can swap out only the 'bad' values for their SI counterparts, which we refer to as a partial swap. We examine both of these options experimentally.

5.3.2 Evaluation

Since we are interested in the effects of negative transfer protection on both games with and without inverted goals, we tested on all of the usual variants of Checkers and Connect Four, as well as Reversed Checkers and Reversed Connect Four (which is just Base Connect Four with inverted goals). Partial swaps were allowed for the first set of trials, but were disabled for all those that followed, for reasons to be discussed shortly.

In the tables that follow, '# swaps' refers to the number of trials in which a full heuristic swap occurred. This can only happen in TI-GMCTS.

Each agent ran as a single-threaded process on a Ubuntu 22.04 machine with a 12th Gen Intel i7-12700 processor.

Game	TI- vs. SI-GMCTS (1-1) time control				TI- vs. SI-GMCTS (2-2) time control			
	Wins	Losses	p-value	# swaps	Wins	Losses	p-value	# swaps
Base	47	51	0.69	2	59	47	0.016**	3
Tall	56	44	0.14	6	52	48	0.38	2
Large	53	47	0.31	23	39	61	0.99	17
Edge	55	41	0.092*	8	48	51	0.66	2
36 Turn	59	66	0.76	4	65	52	0.13	0

Game	TI- vs. SI-GMCTS (5-5) time control			
	Wins	Losses	p-value	# swaps
Base	48	51	0.66	0
Tall	49	51	0.62	0
Large	50	50	0.54	5
Edge	54	41	0.11	0
36 Turn	52	37	0.069*	0

Table 5.5: Results for TI-GMCTS vs. SI-GMCTS on variants of Connect Four, with Base Connect Four as the source game. Partial swapping of TI values **was** allowed. Each condition was run for 100 trials, except for 36 Turn Connect Four, which was run for 200 trials.

5.3.3 Results

We begin by comparing the results of TI-GMCTS versus SI-GMCTS on Connect Four with partial swaps allowed (given by Table 5.5) to those of the same match-up with no partial swaps allowed (given by Table 5.6). We observe that allowing partial swaps results in fewer statistically significant victories over SI-GMCTS, even in Large Connect Four, where they had the highest likelihood of making a positive impact. Since allowing partial swaps is both the more complex and less performant option, we do not use them in the experiments that follow.

We may also compare the results of Table 5.6 to those of Table 5.2 to observe the performance cost associated with running negative transfer protection in this form. Although TI-GMCTS is still mostly winning under the (1-1) and (2-2) time controls that it was previously winning, it has lost some ground to SI-GMCTS. Since Table 5.2 showed transfer to be of benefit in all variants, we can view any full swaps to be false positives, even if there is some amount of heuristic misalignment. This has the most impact on the results for Large Connect Four, which sees the most false positives, and also the largest

	TI- vs. SI-GMCTS (1-1) time control				TI- vs. SI-GMCTS (2-2) time control			
Game	Wins	Losses	p-value	# swaps	Wins	Losses	p-value	# swaps
Base	61	35	0.0052***	4	51	48	0.42	2
Tall	61	40	0.018**	8	59	41	0.044**	3
Large	49	51	0.62	22	52	48	0.38	11
Edge	55	40	0.075*	3	58	42	0.067*	1
36 Turn	80	49	0.0040***	4	78	48	0.0048***	0

	TI- vs. SI-GMCTS (5-5) time control			
Game	Wins	Losses	p-value	# swaps
Base	55	43	0.13	0
Tall	53	47	0.31	1
Large	44	56	0.90	3
Edge	54	41	0.11	0
36 Turn	47	34	0.091*	0

Table 5.6: Results for TI-GMCTS vs. SI-GMCTS on variants of Connect Four, with Base Connect Four as the source game. Partial swapping of TI values **was not** allowed. Each condition was run for 100 trials, except for 36 Turn Connect Four, which was run for 200 trials.

degradation in performance.

We note a trend that a shorter initialization period results in more erroneous full swaps. This is an expected result, since we have previously found that there is more variance in simulation initialization on short time scales, which naturally leads to more false positives. The loss in performance that should result from being handicapped in a larger proportion of games is offset by the typically better performance of TI-GMCTS under short time controls.

Table 5.7 gives the results for TI-GMCTS versus UCT and SI-GMCTS on Checkers variants with full swapping enabled. Again, we may consider any full swaps to be false positives. Compared to Table 5.3, which did not allow any swapping of TI values, we see only minor losses in performance, with the largest representing a loss of statistical significance for Base Checkers at (10-10) time controls. Interestingly, even though Big Board Checkers resulted in a similar number of full swaps as Large Connect Four, we do not see the same corresponding drop in performance. Since Checkers is a longer game being run at longer time controls, it may be the case that the setback caused by an erroneous

	TI-GMCTS vs. UCT (5-5) time control			
Game	Wins	Losses	p-value	# swaps
Base	52	22	0.0003***	3
Torus	89	5	< 0.00001***	6
Big Board	81	11	< 0.00001***	27
Small Board	116	48	0.0007***	5
No Middle	79	12	< 0.00001***	4

	TI-GMCTS vs. SI-GMCTS (5-5) time control				TI-GMCTS vs. SI-GMCTS (10-10) time control			
Game	Wins	Losses	p-value	# swaps	Wins	Losses	p-value	# swaps
Base	54	20	< 0.00001***	9	39	34	0.32	0
Torus	55	26	0.0008***	12	51	20	0.0002***	3
Big Board	52	30	0.0099***	23	52	32	0.019**	14
Small Board	79	43	0.0007***	2	82	54	0.010**	0
No Middle	53	31	0.011**	4	43	37	0.29	0

Table 5.7: Results for TI-GMCTS vs. UCT and SI-GMCTS on variants of Checkers, with negative transfer protection enabled. For each condition, Base Checkers was used as the source game. For the Small Board condition, 200 trials were run. For every other condition, 100 trials were run. '# swaps' refers to the number of trials for which heuristic TI parameters were fully swapped for SI values.

swap is less detrimental than it is in Connect Four. Additionally, a game of Connect Four can end much earlier than a game of Checkers, which places greater importance on the early portion of the game, where the setback is most felt.

Having considered the cases where negative transfer protection can only hurt TI-GMCTS, we now look at those that it is meant to help. Table 5.8 gives results for match-ups between TI-GMCTS, SI-GMCTS, and UCT on Reversed Connect Four (Base Connect Four with its reward values swapped). First, we note that the full swap rate for TI-GMCTS is very high ($\geq 95/100$), which is a desirable outcome. This, combined with the false positive rate for games like Large Connect Four, suggests that the threshold for full swaps could reasonably be increased to preserve performance in the latter.

We next observe that performance is mixed across all match-ups. To highlight this, Table 5.8 always gives p-values from the perspective of the winning agent, and there is only one match-up that shows even weak significance (SI-GMCTS versus UCT at a (5-5) time control, in which SI-GMCTS loses). For the SI-GMCTS versus UCT match-up,

Reversed Connect Four							
Agent	Opponent	Time Control	Wins	Losses	Better?	p-value	# swaps
SI-GMCTS	UCT	(1-1)	53	47	Yes	0.31	-
SI-GMCTS	UCT	(2-2)	53	46	Yes	0.27	-
SI-GMCTS	UCT	(5-5)	40	54	No	0.090*	-
TI-GMCTS	UCT	(1-1)	56	44	Yes	0.14	95
TI-GMCTS	UCT	(2-2)	45	53	No	0.24	95
TI-GMCTS	UCT	(5-5)	45	52	No	0.27	96
TI-GMCTS	SI-GMCTS	(1-1)	44	56	No	0.14	98
TI-GMCTS	SI-GMCTS	(2-2)	55	45	Yes	0.18	96
TI-GMCTS	SI-GMCTS	(5-5)	45	53	No	0.24	99

Table 5.8: Results for Reversed Connect Four with negative transfer detection enabled for TI-GMCTS. For each condition, Base Connect Four was used as the source game (if TI-GMCTS was present) and 100 trials were run. 'Better?' asks whether the first agent won more games than its opponent, and 'p-value' gives the statistical significance of that result. '# swaps' refers to the number of trials for which heuristic TI parameters were fully swapped for SI values.

this continues the trend of SI-GMCTS struggling in Connect Four, reversed goals or not. In cases where a full swap occurs, TI-GMCTS acts as an SI-GMCTS agent with a time penalty, so the performance of SI-GMCTS sets a soft upper bound on the performance of TI-GMCTS. In fact, they appear to perform indistinguishably.

Table 5.9 gives results for TI-GMCTS, SI-GMCTS, and UCT on Reversed Checkers. Here, there is a much clearer hierarchy, in which SI-GMCTS outperforms TI-GMCTS, which outperforms UCT, although the only results of statistical significance are in the SI-GMCTS versus UCT match-up. In Section 4.3.3, when TI-GMCTS had no negative transfer detection, it lost badly to both SI-GMCTS and UCT, with its best record being 11 wins to 76 losses. In comparison, Table 5.9 shows an enormous improvement, and also shows detection to be extremely reliable in Checkers, with a full swap occurring in every trial.

Overall, we find that negative transfer protection performed to the best of our expectations for games with inverted goals. This comes at some cost to other game variants, though the swapping threshold can be adjusted to make this fail-safe less obtrusive.

Reversed Checkers							
Agent	Opponent	Time Control	Wins	Losses	Better?	p-value	# swaps
SI-GMCTS	UCT	(5-5)	46	29	Yes	0.032**	-
SI-GMCTS	UCT	(10-10)	49	23	Yes	0.0023***	-
TI-GMCTS	UCT	(5-5)	41	32	Yes	0.18	100
TI-GMCTS	UCT	(10-10)	43	33	Yes	0.15	100
TI-GMCTS	SI-GMCTS	(5-5)	36	45	No	0.19	100
TI-GMCTS	SI-GMCTS	(10-10)	33	45	No	0.11	100

Table 5.9: Results for Reversed Checkers with negative transfer detection enabled for TI-GMCTS. For each condition, Base Checkers was used as the source game (if TI-GMCTS was present) and 100 trials were run. 'Better?' asks whether the first agent won more games than its opponent, and 'p-value' gives the statistical significance of that result. '# swaps' refers to the number of trials for which heuristic TI parameters were fully swapped for SI values.

5.4 Single-Player Games

To this point, we have been concerned solely with two-player, adversarial games, as is the fashion in GGP. Certainly, single-player games for GGP exist (several of which can be found in the base game repository [1]), but their use is not as well represented in the GGP literature. Shortly, we will discuss some of the properties that can make them undesirable from a research perspective, and may be the reason for this disparity. Even so, we have dedicated this section to single-player games and believe that it is a worthwhile exercise to apply our methods (specifically, TI-GMCTS) to them. This belief is rooted in the ambition for GGP to be useful for more than play.

GGP was established to promote competition, and at the annual GGP tournaments, that competition became a spectacle in which conference-goers watched agents play against each other in real time. Sometimes, competition even extended to human vs. machine show matches, before the machines' victory became a foregone conclusion. In that context, it made sense to evaluate agents on games that a typical person might know well and be able to appreciate as a spectator. Common board games (or variants of them) were well-suited to that purpose, and became staple benchmarks in the GGP literature.

However, GGP has never been limited to the study of 'fun' games. At a high level, a GGP agent is a general-purpose problem solver that can begin taking action almost immediately. That sounds like the description of a revolutionary technology, a robot or personal assistant from science fiction. It sounds very close to artificial general intelligence.

Of course, in focusing from a high level down to the nuts and bolts of implementation, it becomes clear that there remains much work to be done before anything like general intelligence can be claimed. Nevertheless, we should not forget such goals as we take small steps toward them. If we imagine that a GGP agent might one day assist in various ordinary tasks, then we should acknowledge that such tasks are often non-adversarial. According to Kuhn [47], even scientific discovery may be viewed as a sort of puzzle.

This is our motivation for examining single-player games. In the remainder of this section, we will discuss their unique properties, introduce a game for case study, and show that TI-GMCTS can profit from transfer from a two-player source game to single-player target.

5.4.1 Properties of Single-Player Games

Single-player, deterministic games are often referred to as 'puzzles'. Without an opponent to affect the outcome, a sequence of moves made from the initial state will always produce the same result for a puzzle. For example, given the initial grid of a Sudoku, a correct solution will always be correct, no matter how many times the game is repeated. A different initial grid could accommodate a distinct set of solutions, but this would be considered a different game with its own GDL description.

This property is consequential to the way in which we approach single-player games in GGP. In some ways, they are simpler. Without an adversary or non-determinism, solving a puzzle becomes a problem of pure search. If ever a positive reward is found at some terminal node, we know that reward, at minimum, is guaranteed. However, it would be incorrect to say that solving a puzzle is necessarily easier for a GGP agent than playing a two-player game, since we can design puzzles with state spaces that are arbitrarily large. For example, it is much easier to play Tic-tac-toe optimally than it is to 'play a Sudoku' optimally, because a Sudoku's state space is many times larger, and may contain just one correct solution that must be discovered amidst all of that space.

It is also the case that the nature of evaluation changes for single-player games. In two-player games (and, in particular, the zero-sum games that are most common in GGP), an agent must be evaluated relative to its opponent. Even for a game with a vast state space, of which each agent can hope to see no more than a tiny fraction, it is still possible for one agent to distinguish itself from the other by making better use of that fraction. Winning a game does not require optimal play, only play that is better than the opponent's.

On the other hand, we must be more careful in the selection of a single-player game for evaluation. As in two-player games, we must ensure a minimum level of complexity

that prevents agents from fully searching the state space (which would guarantee optimal play and make two agents impossible to differentiate). Additionally, we must ensure that a solution is not so difficult to find that no agent can do so, even after many trials. For the games described in the following section, we have chosen board layouts and time controls such that a UCT agent gives a perfect solution some of the time, but not all of the time.

We must also be careful to ensure that transfer does not trivialize the games. We want a family of games in which the same general strategies can be applied, but not games that actually share solutions. If the source and target game did share solutions (which is guaranteed in the case of self-mapping), it would be trivial to solve the target game by simply trying the solution that worked in the source game at the cost of one rollout. This is only an issue for single-player games because there is no opponent to prevent a game from taking exactly our desired path.

Even if we simply elect not to try source solutions directly, general heuristics cannot be trusted to sufficiently abstract away those solutions on their own. In particular, the specific history heuristic can tell us all of the moves that are part of a solution (because they are associated with non-zero reward) and all of the ones that are not (because they can only lead to zero reward). We observed this to be the case for two variants of the Eight Queens puzzle in the base GGP games repository. In one variant, only legal moves are allowed, which very quickly reduces the game's branching factor and makes the task of finding a solution manageable in a reasonable time frame. In the other variant, all possible moves are allowed, including those that violate the rules and therefore make a solution impossible. After making such a move, an agent simply continues to play, and only receives a score of zero once all queens have been placed. Taken on its own, UCT was not able to solve this variant. However, we found that it could be solved trivially with knowledge transferred from the legal-moves-only variant, for which solutions were known.

Although this behaviour is to the benefit of a transfer agent, it does not make for compelling experimentation. In the sections that follow, we avoid any possibility of it by transferring not between two single-player games, but from a two-player game to one that is single-player.

5.4.2 Vacuum Checkers

Vacuum Checkers is a single-player variant of Checkers that we designed to test transfer from a two-player game (in this case, Base Checkers) to a single-player puzzle. In particular, we wanted to be able to create a collection of games with the same dynamics (with one

exception, to be described shortly), but different initial board layouts that make transfer more or less effective. The rules of Vacuum Checkers are as follows:

> The player begins with a single red king somewhere on the board. This piece moves identically to a king in Base Checkers (diagonally, in any direction). There is also some number of black pieces on the board. These pieces do not move because Red is the only player.
>
> The red king may capture black pieces as it would in Base Checkers. (That is, when it is diagonally adjacent to a black piece and there is an open space on the other side, it may jump the piece, removing that piece from the board and moving itself to the aforementioned empty space.) Double and triple jumps are permitted when captures can be chained consecutively. These are still considered one move.
>
> The objective of the game is for the red king to capture all of the black pieces in the minimum number of moves possible.

Vacuum Checkers is a path-finding puzzle on a Checkers board. It is named for its similarity to the mundane task of vacuum cleaning. The red king is the vacuum cleaner, and it must run over piles of dirt to remove them. Although double and triple jumps stretch this analogy, we have preserved them in the rules in order to observe the effect of transfer when they are prominent features of the optimal path.

We consider three board layouts of Vacuum Checkers:

- Layout A: Seven black pieces are positioned to force alternating captures and single-square movement. One double jump must be made toward the end of the optimal path, which is 11 moves long.
- Layout B: Eight black pieces are placed such that all may be captured via double jumps, alternating with single-square movement. The optimal path is 7 moves long.
- Layout C: Three pieces are positioned near the corners of the board that are farthest from the red king. Many single-square movements are necessary between captures. An optimal path is 12 moves long.

Figure 5.2 depicts these layouts, as well as their optimal solutions.

In general, we expected that some, but not all of the general heuristics imported from Base Checkers would be useful. In both games, capturing pieces is key to winning, so

heuristics like the history heuristic for double jumping and the number of black piece symbols were identified as likely to be useful. On the other hand, the mobility heuristic (the difference between the number of moves available to each player), which produced strong correlation values for Base Checkers, becomes meaningless in Vacuum Checkers. Other heuristics, such as the number of red pieces, also become unhelpful because they do not change between states. Additionally, TI-GMCTS must always begin at a search deficit because of its relatively lengthy initialization process.

Our three board layouts were designed to see if the transferred knowledge could overcome these drawbacks when different types of moves (and by extension, general heuristics) were emphasized. Layout A consists of many individual captures, creating opportunity for the black piece count to be useful. Layout B emphasizes the importance of double jumps, which supports the history and symbol count heuristics. Layout C offers very few captures and no double jumps. This was the hardest case for useful transfer, and gave the largest advantage to pure, cheap search. Note that jumps are not obligatory, as they would be in some Checkers rule sets.

For each board layout, we created two separate games whose rules differed only in their reward structure. In the base version (which we refer to with no affixes), finding a path of minimum length rewards 100 points, but 20 points are lost for each additional move used beyond that minimum. In the 'sparse' variants, 100 points are awarded for a path of minimum length, and no points are awarded for longer paths. In all cases, games terminate when all black pieces have been captured, or when 20 moves have been made.

5.4.3 Evaluation

For this set of experiments, we ran only TI-GMCTS and UCT because simulation initialization does not make sense in the context of single-player puzzles. In order to get any positive reward during initial simulations (which is necessary for weighting the heuristics), a solution would have to be found, thereby negating the need for the use of heuristics. Similarly, negative transfer protection for TI-GMCTS cannot be applied, and was disabled. All experiments involving TI-GMCTS use two-player Base Checkers as the source game for transfer.

Each trial was performed as a single-threaded Java process on a Windows 10 machine with an Intel i7-7700HQ processor. This processor, which is weaker than was used for previous experiments, enables us to observe meaningful differences in performance at different time controls without needing to use increments of less than one second.

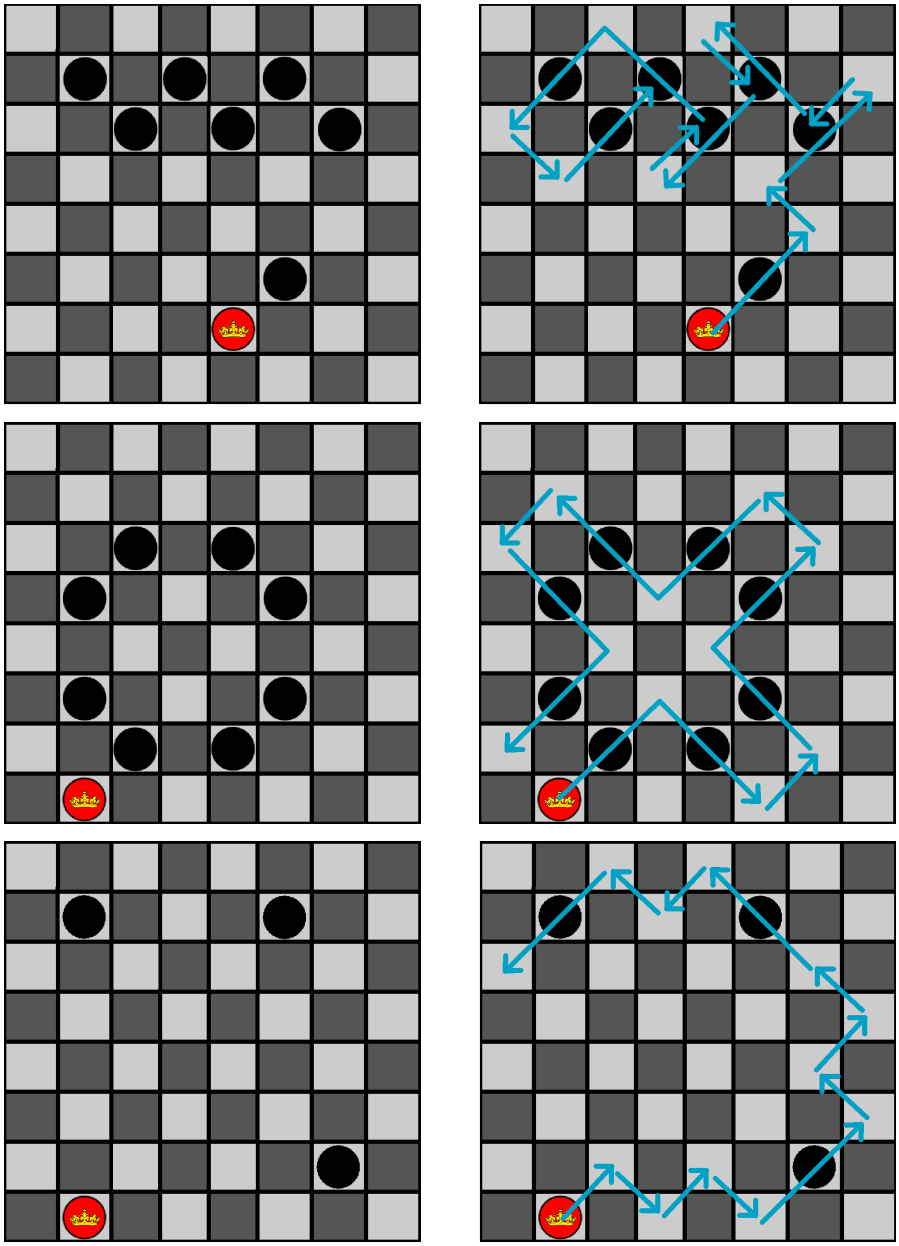

Figure 5.2: Vacuum Checkers starting positions on the left, with their corresponding optimal solutions on the right. *Top*: Layout A, which emphasizes single-piece captures. *Middle*: Layout B, which emphasizes double jumps. *Bottom*: Layout C, which has few captures and an optimal solution that keeps near the edges of the board.

Game	TI-GMCTS (2-2) time control		UCT (2-2) time control	
	Avg. Score	Perfect Wins	Avg. Score	Perfect Wins
Layout A	**100.0**	100	39.8	28
Layout B	**100.0**	100	41.0	17
Layout C	77.6	0	**86.2**	58
Layout A - Sparse	**100.0**	100	9.0	9
Layout B - Sparse	**100.0**	100	24.0	24
Layout C - Sparse	0	0	**24.0**	24

Game	TI-GMCTS (3-3) time control		UCT (3-3) time control	
	Avg. Score	Perfect Wins	Avg. Score	Perfect Wins
Layout A	**100.0**	100	69.8	51
Layout B	**100.0**	100	67.6	40
Layout C	69.8	4	**92.0**	67
Layout A - Sparse	**100.0**	100	17.0	17
Layout B - Sparse	**100.0**	100	10.0	10
Layout C - Sparse	0.0	0	**31.0**	31

Table 5.10: Results for TI-GMCTS and UCT on variants of Vacuum Checkers at (2-2) and (3-3) time controls, with Base Checkers as the source game, for 100 trials each. The best average score for each variant is in bold.

5.4.4 Results

Table 5.10 gives results for TI-GMCTS and UCT on our six variants of Vacuum Checkers. The performance of UCT gives us some insight into the difficulties of these various conditions from the perspective of pure search. Layouts A and B are similar in that they have unique optimal paths. Although Layout A's path is a little longer, Layout B compensates with the extra branching factor caused by the availability of double jumps. As a result, UCT achieves a similar level of performance on each. Even though Layout C has the longest optimal path, that path is not unique, which provides more opportunities to find a solution, even when searching blindly. UCT performs its best on this layout.

As expected, we also see a decrease in the performance of UCT when less time is given, and when partial rewards are withheld. Notably, the latter is not due only to the lack of the rewards themselves, but also to the information that they provide. In all cases but one, the number of perfect wins also decreases sharply for the sparse variants.

TI-GMCTS excelled in the conditions that catered to its general heuristics (Layouts

A and B), and struggled in the one that did not. This was particularly evident in the sparse version of Layout C, which TI-GMCTS was not able to solve a single time. When we examine the cost of initializing transfer, we see why this is the case. TI-GMCTS took an average of 1025 ms to initialize, while UCT took an average of only 22 ms. At time controls this short, that represents a significant difference in the amount of search that can be done before the first move must be played. This is an issue that compounds with the work required to calculate heuristics, as well as the fact that later rollouts are faster than earlier ones, because more of the MCT is already constructed for them. At a (3-3) time control, TI-GMCTS had completed an average of 1692 rollouts by the time it had to play its first move, while UCT had completed an average of 6678 rollouts.

On the other hand, we observe from the performance of TI-GMCTS on Layouts A and B that when guidance is effective, it can lead to a solution very consistently (in this case, every time). If not for the single-square moves in between jumps, a solution would have been found on the first rollout. Solutions were found so quickly that there was no gap in performance made by withholding partial rewards or using a shorter time control (which cannot be reduced further while still allowing TI-GMCTS to complete inialization).

We conclude that transfer from a two-player game to a single-player game is possible and useful when the means of reaching their objectives are sufficiently aligned. In both Base and Vacuum Checkers, the game is won by reducing the number of opposing pieces, and utilizing this information led to a more well-directed use of resources for the layouts of Vacuum Checkers in which many captures were available.

This has useful implications for transfer to single-player games in general. Finding useful heuristics can be difficult in single-player games with sparse rewards because failure means no positive reward was ever discovered. Heuristics cannot be meaningfully initialized if every move and every board position led to zero reward. On the other hand, in two-player (zero sum) games, someone has to win. An agent can learn without first needing to find a needle in a haystack. If that knowledge can be transferred to a hard single-player game, it could represent a better investment of time than simply searching the game itself.

Chapter 6

Conclusion

General Game Playing is a field of research in which one agent must play many games. All GGP research recognizes this as a challenge, but relatively little also sees it as an opportunity. In this thesis, we have developed methods that enable an agent to profit from past experience in games that it has seen before, with the potential to grow in knowledge as its library expands. In summary, our contributions are as follows.

We developed two algorithms, LMap and MMap, for finding a mapping between the most similar symbols in two different, but related games. We showed their accuracy to be better than a baseline mapper, and further demonstrated their usefulness as an integral part of our agents in all of the work that followed.

We created a full system for guiding MCTS with transferred knowledge (TI-GMCTS). This system includes methods for storing and organizing data, a modular system for calculating general heuristics, and an updated version of UCB1 that weights the importance of heuristics based on the strength of their correlation with reward. We showed that, when time for initialization is short, TI-GMCTS performs better than both its simulation-initialized counterpart (SI-GMCTS), and a UCT baseline on two families of game variants.

Finally, we addressed source game selection, the last missing piece of a full GGP agent, and showed various ways that TI-GMCTS can be made to apply more generally, through the addition of new kinds of heuristics (board heuristics), new kinds of games (single-player), and a fail-safe for negative transfer detection.

6.1 Future Work

Source Game Selection: Each of the topics discussed in Chapter 5 represents the tip of an iceberg for possible future research. This is most explicit in Section 5.1, which discusses the selection of a source game, and acknowledges that a full treatment should address the question of how large a library of known games is reasonable to keep. This requires striking a balance between the difficulty in finding the best source game from among many in a large library, and having that source game be a good match for the current target game. It is a difficult question to answer because there is not one distribution of possible target games that we can query. In principle, it is always possible to create a totally new game that is not similar to any seen before, but practically, there are a finite number of game archetypes that are varied and recycled in GGP. Ultimately, it is likely that an arbitrary set needs to be designated, and the collection of standard (i.e. non-variant) games in the GGP base repository seems a reasonable choice.

There are also possibilities for technical improvements in source game selection. For one, LMap may be modified specifically for the task of finding distances efficiently. 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 varying the maximum search depth, balancing the competing interests of speed and accuracy, particularly if it were necessary to adopt multiple elimination rounds.

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 would also provide some relief to the problem of games that are the same, but are described differently enough to inhibit a good mapping. We would not be able to recognize the similarity of the descriptions, but could store all of them within the hierarchy without significantly hampering execution time.

Negative Transfer Protection: In Section 5.3, we developed a method for detecting negative transfer in individual heuristic values, and swapping them for simulation-initialized (SI) values. However, we ultimately found that the noisiness of the SI values resulted in errors that hurt performance compared to the more conservative approach of allowing only full swaps of all heuristic values. This is an idea that warrants more research, as the ability to specifically filter out a few bad heuristics from an otherwise useful source

game (like the history heuristic in Large Connect Four) is an attractive prospect.

Modularity: Section 5.2 demonstrates the modularity of the set of heuristics, and shows how loosening our restrictions on their generality can open the door to incorporating new sets of them. In future work, it would be interesting accept even more specific heuristics to observe their effect on performance. For example, incorporating notions of piece ownership permits heuristics like the distance between opposing pieces, or the area of the board controlled.

Dealing with modularity more broadly, linear regression was a simple, lightweight choice for finding a correlation between heuristic values and expected reward. It would be possible to drop in other methods of calculating a heuristic evaluation function, under the constraint that they must not use up too much of the start clock (if we still want to allow simulation initialization, or features like negative transfer detection). For example, it could be possible to take simple machine learning approaches, like a support vector machine, random forest, or even a basic neural network.

Even Greater Generality: As mentioned in the introduction to Chapter 5, the pursuit of generality is a perpetual challenge, and there are natural extensions of this work in furtherance of that goal. A simple avenue is expansion into other games, as was done in Section 5.4. In this thesis, we have kept our selection limited to three to maintain a manageable scope, since each game has at least four variants (plus the original). We selected games from the base repository that already had at least a few variants available, indicating some level of interest within the broader GGP community. We would want any additional games to also be amenable to several variations, and to explore some new aspect of the space of possible games. A non-board game, like Nim, could be an interesting test bed, although some care would need to be taken in ensuring a high enough level of complexity to be interesting.

Branching out a little further, our methods are agnostic to non-determinism, although the version of the GGP library that we have used does not support it. Incorporating Thielscher's GDL extension [86] would permit non-deterministic games (and indeed, any extensive-form game [87]). With it, dice games like Backgammon, and card games like Poker become accessible for testing.

Further still, we are interested in adapting our mapping methods to languages other than GDL. In particular, game specifications in Ludii [61] are also naturally represented as graphs, albeit with many more colours than the rule graphs of GDL. There is already

interest in general heuristics [74] and conducting transfer [72] under the Ludii framework, which provides a natural place for our work to fit in.

6.2 Closing Thoughts

With the last Annual GGP Competition being held in 2016, it now falls to individual researchers to carry the torch. Without an incentive to pursue those methods that provide the most competitive advantage, there is opportunity to explore the ideas that lie on the periphery of GGP. Transfer is one such idea. Though we (and others) have contributed tools and ideas, transfer is still in its infancy in this research area. There is still great untapped potential. There is still so much work left to do.

If the GGP community is to remain fractured going forward, then it is our prerogative and responsibility to choose which aspects of it to keep, and which to push in new directions. For some, that means developing new, expressive game languages. For others, it means relaxing time restrictions to allow for methods like deep learning. We hope that some, like us, will see the value in pushing in the opposite direction toward ever shorter time scales. Doing more with less is well in keeping with the spirit of GGP. It is a torch worth carrying.

References

[1] Ggp.org games base repository. http://games.ggp.org/base/, 2023.

[2] David Abel, Dilip Arumugam, Lucas Lehnert, and Michael Littman. State abstractions for lifelong reinforcement learning. In *International Conference on Machine Learning*, pages 10–19, 2018.

[3] David Abel, Dilip Arumugam, Lucas Lehnert, and Michael L Littman. Toward good abstractions for lifelong learning. In *Proceedings of the NIPS Workshop on Hierarchical Reinforcement Learning*, 2017.

[4] David Abel, Yuu Jinnai, Sophie Yue Guo, George Konidaris, and Michael Littman. Policy and value transfer in lifelong reinforcement learning. In *International Conference on Machine Learning*, pages 20–29, 2018.

[5] Louis Victor Allis et al. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.

[6] Haitham Bou Ammar, Eric Eaton, Matthew E Taylor, Decebal Constantin Mocanu, Kurt Driessens, Gerhard Weiss, and Karl Tuyls. An automated measure of mdp similarity for transfer in reinforcement learning. In *Workshops at the Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

[7] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2-3):235–256, 2002.

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

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

[10] Marc Bellemare, Joel Veness, and Michael Bowling. Bayesian learning of recursively factored environments. In *International Conference on Machine Learning*, pages 1211–1219, 2013.

[11] Reinaldo AC Bianchi, Luiz A Celiberto Jr, Paulo E Santos, Jackson P Matsuura, and Ramon Lopez de Mantaras. Transferring knowledge as heuristics in reinforcement learning: A case-based approach. *Artificial Intelligence*, 226:102–121, 2015.

[12] Emma Brunskill and Lihong Li. Pac-inspired option discovery in lifelong reinforcement learning. In *International conference on machine learning*, pages 316–324, 2014.

[13] Tim Brys, Anna Harutyunyan, Matthew E Taylor, and Ann Nowé. Ensembles of shapings.

[14] Tim Brys, Anna Harutyunyan, Matthew E Taylor, and Ann Nowé. Policy transfer using reward shaping. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, pages 181–188. International Foundation for Autonomous Agents and Multiagent Systems, 2015.

[15] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

[16] Brooke Chan. Openai five, Jun 2019.

[17] James Clune. Heuristic evaluation functions for general game playing. In *AAAI*, volume 7, pages 1134–1139, 2007.

[18] Stephen A Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, 1971.

[19] Eyal Even-Dar and Yishay Mansour. Approximate equivalence of markov decision processes. In *Learning Theory and Kernel Machines*, pages 581–594. Springer, 2003.

[20] Anestis Fachantidis, Ioannis Partalas, Matthew E Taylor, and Ioannis Vlahavas. Transfer learning via multiple inter-task mappings. In *European Workshop on Reinforcement Learning*, pages 225–236. Springer, 2011.

[21] Brian Falkenhainer, Kenneth D Forbus, and Dedre Gentner. The structure-mapping engine: Algorithm and examples. *Artificial intelligence*, 41(1):1–63, 1989.

[22] Fernando Fernández and Manuela Veloso. Policy reuse for transfer learning across tasks with different state and action spaces. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning*. Citeseer, 2006.

[23] Fernando Fernández and Manuela Veloso. Probabilistic policy reuse in a reinforcement learning agent. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 720–727. ACM, 2006.

[24] Hilmar Finnsson and Yngvi Björnsson. Cadiaplayer: Search-control techniques. *KI-Künstliche Intelligenz*, 25:9–16, 2011.

[25] Hilmar Finnsson and Yngvi Björnsson. Game-tree properties and mcts performance. In *IJCAI*, volume 11, pages 23–30, 2011.

[26] Hervé Gallaire and Jack Minker. Logic and data bases, symposium on logic and data bases, centre d'études et de recherches de toulouse, 1977. *Advances in Data Base Theory*, 1978.

[27] Michael Genesereth and Yngvi Björnsson. The international general game playing competition. *AI Magazine*, 34(2):107–107, 2013.

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

[29] Michael Genesereth and Michael Thielscher. General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(2):1–229, 2014.

[30] Juris Hartmanis. Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson). *Siam Review*, 24(1):90, 1982.

[31] Anna Harutyunyan, Sam Devlin, Peter Vrancx, and Ann Nowé. Expressing arbitrary reward functions as potential-based advice. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[32] Sebastian Haufe, Daniel Michulke, Stephan Schiffel, and Michael Thielscher. Knowledge-based general game playing. *KI-Künstliche Intelligenz*, 25(1):25–33, 2011.

[33] Thomas Hinrichs and Kenneth D Forbus. Transfer learning through analogy in games. *Ai Magazine*, 32(1):70–70, 2011.

[34] Aline Hufschmitt, Jean-Noël Vittaut, and Jean Méhat. A general approach of game description decomposition for general game playing. In *Computer Games*, pages 165–177. Springer, 2016.

[35] Guifei Jiang, Laurent Perrussel, Dongmo Zhang, Heng Zhang, and Yuzhi Zhang. Game equivalence and bisimulation for game description language. In *Pacific rim intl. conf. on artificial intelligence*, pages 583–596. Springer, 2019.

[36] Joshua DA Jung and Jesse Hoey. Distance-based mapping for general game playing. In *2021 IEEE Conference on Games (CoG)*, pages 1–8. IEEE, 2021.

[37] David 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*, pages 1–7, 2007.

[38] Lukasz Kaiser, Mohammad Babaeizadeh, Piotr Milos, Blazej Osinski, Roy H Campbell, Konrad Czechowski, Dumitru Erhan, Chelsea Finn, Piotr Kozakowski, Sergey Levine, et al. Model-based reinforcement learning for atari. *arXiv preprint arXiv:1903.00374*, 2019.

[39] Mesut Kirci, Nathan Sturtevant, and Jonathan Schaeffer. A ggp feature learning algorithm. *KI-Künstliche Intelligenz*, 25(1):35–42, 2011.

[40] ww Klenk and Ken Forbus. Cross domain analogies for learning domain theories. Technical report, Northwestern Univ, Evanston, IL, Qualitative Reasoning Group, 2007.

[41] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[42] George Konidaris and Andrew G Barto. Building portable options: Skill transfer in reinforcement learning. In *IJCAI*, volume 7, pages 895–900, 2007.

[43] Gregory Kuhlmann and Peter Stone. Automatic heuristic construction for general game playing. In *AAAI*, pages 1883–1884, 2006.

[44] Gregory Kuhlmann and Peter Stone. Automatic heuristic construction in a complete general game player. In *AAAI*, volume 6, pages 1457–62, 2006.

[45] Gregory Kuhlmann and Peter Stone. Graph-based domain mapping for transfer learning in general games. In *European Conference on Machine Learning*, pages 188–200. Springer, 2007.

[46] Gregory John Kuhlmann. *Automated domain analysis and transfer learning in general game playing*. PhD thesis, Uni. of Texas at Austin, 2010.

[47] Thomas S Kuhn. *The structure of scientific revolutions*. University of Chicago press, 2012.

[48] Alessandro Lazaric. *Knowledge transfer in reinforcement learning*. PhD thesis, PhD thesis, Politecnico di Milano, 2008.

[49] Lihong Li, Thomas J Walsh, and Michael L Littman. Towards a unified theory of state abstraction for mdps. In *ISAIM*, 2006.

[50] Sili Liang, Guifei Jiang, and Yuzhi Zhang. Combining m-mcts and deep reinforcement learning for general game playing. In *International Conference on Distributed Artificial Intelligence*, pages 221–234. Springer, 2021.

[51] Yaxin Liu and Peter Stone. Value-function-based transfer for reinforcement learning using structure mapping. In *Proceedings of the national conference on artificial intelligence*, volume 21, page 415. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[52] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. 2008.

[53] Jacek Mańdziuk and Maciej Świechowski. Generic heuristic approach to general game playing. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 649–660. Springer, 2012.

[54] Daniel Michulke. Heuristic interpretation of predicate logic expressions in general game playing. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'11)*, pages 61–68. Citeseer, 2011.

[55] Daniel Michulke and Stephan Schiffel. Admissible distance heuristics for general games. In *International Conference on Agents and Artificial Intelligence*, pages 188–203. Springer, 2012.

[56] Daniel Michulke and Stephan Schiffel. Distance features for general game playing agents. In *ICAART (1)*, pages 127–136, 2012.

[57] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[58] Andrew Y Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, pages 278–287, 1999.

[59] Barney Pell. Metagame in symmetric chess-like games. 1992.

[60] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D Gaina, Julian Togelius, and Simon M Lucas. General video game ai: a multi-track framework for evaluating agents, games and content generation algorithms. *arXiv preprint arXiv:1802.10363*, 2018.

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

[62] Michel Quenault and Tristan Cazenave. „extended general gaming model “. In *Computer Games Workshop*, pages 195–204, 2007.

[63] Kaspar Riesen, Miquel Ferrer, and Horst Bunke. Approximate graph edit distance in quadratic time. *IEEE/ACM transactions on computational biology and bioinformatics*, 2015.

[64] et al. Sam Schreiber, Alex Landau. The general game playing base package, 2018.

[65] Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21–21, 1996.

[66] Tom Schaul. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8. IEEE, 2013.

[67] Stephan Schiffel and Michael Thielscher. Automatic construction of a heuristic search function for general game playing. *Department of Computer Science*, pages 16–17, 2006.

[68] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Aaai*, volume 7, pages 1191–1196, 2007.

[69] Max Schwarzer, Johan Obando-Ceron, Aaron Courville, Marc Bellemare, Rishabh Agarwal, and Pablo Samuel Castro. Bigger, better, faster: Human-level atari with human-level efficiency. *arXiv preprint arXiv:2305.19452*, 2023.

[70] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[71] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[72] Dennis JNJ Soemers, Vegard Mella, Eric Piette, Matthew Stephenson, Cameron Browne, and Olivier Teytaud. Transfer of fully convolutional policy-value networks between games and game variants. *arXiv preprint arXiv:2102.12375*, 2021.

[73] Vishal Soni and Satinder Singh. Using homomorphisms to transfer options across continuous reinforcement learning domains. In *AAAI*, volume 6, pages 494–499, 2006.

[74] Matthew Stephenson, Dennis JNJ Soemers, Éric Piette, and Cameron Browne. General game heuristic prediction based on ludeme descriptions. In *2021 IEEE Conference on Games (CoG)*, pages 1–4. IEEE, 2021.

[75] Alexander L Strehl. *Probably approximately correct (PAC) exploration in reinforcement learning*. PhD thesis, Rutgers University-Graduate School-New Brunswick, 2007.

[76] Richard S Sutton, Doina Precup, and Satinder P Singh. Intra-option learning about temporally abstract actions. In *ICML*, volume 98, pages 556–564, 1998.

[77] Richard S Sutton, Doina Precup, and Satinder P Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[78] Maciej Świechowski and Jacek Mańdziuk. Self-adaptation of playing strategies in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):367–381, 2013.

[79] Maciej Świechowski, HyunSoo Park, Jacek Mańdziuk, and Kyung-Joong Kim. Recent advances in general game playing. *The Scientific World Journal*, 2015, 2015.

[80] Mandy JW Tak, Mark HM Winands, and Yngvi Bjornsson. N-grams and the last-good-reply policy applied in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):73–83, 2012.

[81] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, and Chunfang Liu. A survey on deep transfer learning. In *Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part III 27*, pages 270–279. Springer, 2018.

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

[83] Matthew E Taylor, Gregory Kuhlmann, and Peter Stone. Autonomous transfer for reinforcement learning. In *AAMAS (1)*, pages 283–290. Citeseer, 2008.

[84] Matthew E Taylor and Peter Stone. Behavior transfer for value-function-based reinforcement learning. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 53–59. ACM, 2005.

[85] Matthew E Taylor and Peter Stone. An introduction to intertask transfer for reinforcement learning. *Ai Magazine*, 32(1):15–15, 2011.

[86] Michael Thielscher. A general game description language for incomplete information games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 24, pages 994–999, 2010.

[87] Michael Thielscher. The general game playing description language is universal. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1107. Citeseer, 2011.

[88] Ruben Rodriguez Torrado, Philip Bontrager, Julian Togelius, Jialin Liu, and Diego Perez-Liebana. Deep reinforcement learning for general video game ai. In *2018 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 1–8. IEEE, 2018.

[89] Lisa Torrey, Jude Shavlik, Trevor Walker, and Richard Maclin. Skill acquisition via transfer learning and advice taking. In *European Conference on Machine Learning*, pages 425–436. Springer, 2006.

[90] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, et al. Alphastar: Mastering the real-time strategy game starcraft ii. *DeepMind Blog*, 2019.

[91] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.

[92] Karol Walędzik and Jacek Mańdziuk. An automatically generated evaluation function in general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(3):258–270, 2013.

[93] Thomas J Walsh, Lihong Li, and Michael L Littman. Transferring state abstractions between mdps. In *ICML Workshop on Structural Knowledge Transfer for Machine Learning*, 2006.

[94] Eric Wiewiora, Garrison W Cottrell, and Charles Elkan. Principled methods for advising reinforcement learning agents. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 792–799, 2003.

[95] Chenjun Xiao, Jincheng Mei, and Martin Müller. Memory-augmented monte carlo tree search. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.