Combining M-MCTS and Deep Reinforcement Learning for General Game Playing

Sili Liang, Guifei Jiang(✉), and Yuzhi Zhang

College of Software, Nankai University, Tianjin 300350, China
2120210551@mail.nankai.edu.cn, {g.jiang,zyz}@nankai.edu.cn

Abstract. As one of the main research areas in AI, General Game Playing (GGP) is concerned with creating intelligent agents that can play more than one game based on game rules without human intervention. Most recent work has successfully applied deep reinforcement learning to GGP. This paper continues this line of work by integrating the Memory-Augmented Monte Carlo Tree Search algorithm (M-MCTS) with deep reinforcement learning for General Game Playing. We first extend M-MCTS from playing the single game Go to multiple concurrent games so as to cater to the domain of GGP. Then inspired by Goldwaser and Thielscher (2020), we combine the extension with deep reinforcement learning for building a general game player. Finally, we have tested this player on several games compared with the benchmark UCT player, and the experimental results have confirmed the feasibility of applying M-MCTS and deep reinforcement learning to GGP.

Keywords: General Game Playing · Memory-Augmented Monte Carlo Tree Search · Deep reinforcement learning · UCT algorithm · General game player

1 Introduction

In the past two decades, several remarkable results about computer games have been achieved in Artificial Intelligence (AI) [3,13,16,18,19]. These results indicate the incredible improvements of technologies and algorithms in AI. On the other hand, as Feng-hsiung Hsu pointed out, each of these intelligent systems is designed for a specific game, and building systems to play specific games has limited value in AI [13].

To address this issue, a project called General Game Playing (GGP) has been launched at Stanford University since 2005. This project is concerned with creating intelligent systems that understand the rules of arbitrary new games and learn to play these games without human intervention [9]. Unlike specialised systems, a general game player cannot use an algorithm with specific knowledge or heuristics that do not apply to other games [10]. Such a system rather requires

J. Chen et al. (Eds.): DAI 2021, LNAI 13170, pp. 221–234, 2022.
https://doi.org/10.1007/978-3-030-94662-3_14

a form of general intelligence that enables it to autonomously play a new game based on its rules. After more than ten years of development, GGP has become an important research area in AI [22].

Designing and implementing strategy generation algorithms is a core technique in GGP. Nowadays, the main algorithms used in successful general game players are the Monte Carlo Tree Search (MCTS) algorithm and its variants. The key idea of MCTS is to construct a search tree of states evaluated by fast Monte Carlo simulations. As a general-purpose algorithm, MCTS is widely used in GGP. However, the main disadvantage of MCTS is that it makes very limited use of game-related knowledge, which may be inferred from a game description and play a part for game playing [21].

On the other hand, with the great success of AlphaGo and its successors, the role of reinforcement learning in game playing has become prominent [18–20], and several reinforcement learning approaches have been proposed to improve MCTS. For instance, a deep neural network can be used to learn domain knowledge and evaluate a state so as to guide the MCTS search [18]. In particular, an algorithm called Memory-Augmented Monte Carlo Tree Search (M-MCTS) was proposed by incorporating MCTS with a memory structure to exploit generalization in online real-time search. It showed that M-MCTS is useful for improving the performance of MCTS in both theory and practice [24].

Recently there have been works applying reinforcement learning approaches to the domain of GGP [2,7,11,12,23]. A project called RL-GGP was proposed by [2] to integrate GGP with RL-glue so as to compare reinforcement learning algorithms in the context of games. It only provided a framework and did not implement a general game player. In [23], the canonical reinforcement learning method Q-learning was used in GGP. Yet it showed that this method converges much slower than MCTS. Most recently, deep reinforcement learning was applied to GGP by extending the AlphaZero algorithm, and it showed that this can provide competitive results in several games except for a cooperative game Babel [11]. Inspired by this work, GGPZero was developed by [12] to further explore the learning architecture for GGP, and systematic experiments were done to confirm its feasibility. Our work continues this line of work by combining M-MCTS and deep reinforcement learning for GGP and shows that it can outperform the benchmark player in a variety of games, which has further confirmed the feasibility of applying deep reinforcement learning to GGP.

The rest of this paper is structured as follows: Sect. 2 provides the background including GGP, M-MCTS and deep reinforcement learning involved in this paper. Section 3 extends M-MCTS to cater to the domain of GGP and further combines the extension with deep reinforcement learning for building a general game player. Section 4 implements and evaluates the general game player with the benchmark UCT player in terms of various games. Finally, we conclude this paper and point out some future work.

2 Background

In this section, let us provide some preliminaries. We begin with an overview of GGP, and then introduce the Memory-Augmented Monte Carlo Tree Search algorithm and deep reinforcement learning we use for building a general game player.

2.1 General Game Playing

There are many strategy generation algorithms used in previous successful general game players. At the early stage, GGP players mainly took evaluation-based search approach and were mostly based on MiniMax algorithm and its variants. The typical examples are the GGP competition champions Cluneplayer and Fluxplayer [4,17]. Since the 2007 champion Cadiaplayer first adopted the upper confidence bound on trees (UCT) algorithm [6], the simulation-based search approach has gradually become the mainstream algorithm for building a GGP player. All the following winners such as Ary, TurboTurtle used either UCT or its variants [15], except the 2016 winner WoodStock, which took a method based on constraint satisfaction programming [14].

UCT is a game tree search algorithm by combining the Monte Carlo Tree Search method with the Upper Confidence Bound (UCB) [1]. The key idea of UCT is to construct a search tree of states evaluated by fast Monte Carlo simulations, and use the UCB formula to determine whether to continue the search or consider the possibility of other sibling nodes. Each node of the search tree s stores three parameters: a node count $N(s)$, a count of the number of times action a has been taken from node s $N(s,a)$, and a value for how good the current state is, $\hat{V}(s)$. The choice of action at a state is then determined by the following UCB formula [1].

$$\mathbf{argmax}_{a \in A(s)} \left(\hat{V}(\delta(s,a)) + C\sqrt{\frac{ln(N(s))}{N(\delta(s,a))}} \right)$$

where $A(s)$ denotes the set of all legal actions at state s, $\delta(s,a)$ is the state transition function, i.e., the next state after taking action a at state s, and C is a constant used for balancing exploration and exploitation. The higher the constant C, the more UCT will explore unpromising nodes.

2.2 Memory-Augmented Monte Carlo Tree Search

With large state spaces and relatively limited search time, the inaccurate value estimation of a state can mislead building the search tree and degrade the performance of UCT algorithm. To address this issue, the Memory-Augmented Monte Carlo Tree Search algorithm (M-MCTS) was proposed by Xiao *et al.* in [24]. M-MCTS is inspired from the core idea of neural network algorithms: generalization, that is, similar states can share information. Specifically, it exploited

generalization in online real-time search by incorporating MCTS with a memory structure, where each entry contains information of a particular state. This memory is used to generate an approximate value estimation by combining the estimations of similar states, which allows UCT algorithm to improve the accuracy of value estimation.

The approximate value estimation is performed as follows: given a memory $\mathcal{M}$ and a state s, they find the set of most similar states $\mathrm{M}_s \subset \mathcal{M}$ according to a distance metric $d(.,s)$, and compute a memory-based value estimation $\hat{V}_{\mathcal{M}}(s) = \sum_{i=1}^{\mathrm{M}} w_i(s)\hat{V}(i)$ such that $\sum_{i=1}^{\mathrm{M}} w_i(s) = 1$, where $\hat{V}(i)$ denotes the value estimation of state i from simulations, and $w_i(s)$ is called the weight of state i with respect to s.

Each entry of $\mathcal{M}$ corresponds to one particular state $s \in S$. It contains the state's feature representation $\phi(s)$ as well as its simulation statistics $\hat{V}(s)$ and $N(s)$. To integrate memory with MCTS, in each node of an M-MCTS search tree, they store an extended set of statistics

$$\{N(s), \hat{V}(s), N_{\mathcal{M}}(s), \hat{V}_{\mathcal{M}}(s)\}$$

where $N_{\mathcal{M}}(s)$ is the number of evaluations of the approximated memory value $\hat{V}_{\mathcal{M}}(s)$. Then in the action selection policy of UCT, the value $\hat{V}(s)$ of state s in UCB formula is replaced by $(1-\lambda_s)\hat{V}(s) + \lambda_s \hat{V}_{\mathcal{M}}(s)$, where λ_s is the learning rate, a constant parameter to guarantee no bias asymptotically. This process is shown in Fig. 1. It has been shown that the memory based value approximation is better than the vanilla Monte Carlo estimation with high probability under mild conditions, and M-MCTS outperforms the original MCTS with the same number of simulations in the game Go based on a handcrafted neural network [24].

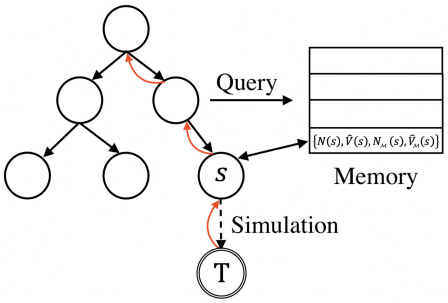

Fig. 1. An illustration of the architecture of M-MCTS, image based on [24]

2.3 Deep Reinforcement Learning

In AlphaGo, the tree search evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play [18].

This program was then upgraded to AlphaGo Zero which is trained solely by self-play reinforcement learning, starting from random play, without any supervision or use of human data [20]. Recently it has been generalized to AlphaZero that achieves superhuman performance in multiple games such as Go, Chess and Shogi by using a general-purpose reinforcement learning algorithm [19].

Specifically, AlphaZero uses a deep neural network which takes a board position as an input, and outputs a probability distribution over moves and an estimation of the expected outcome from the board position. AlphaZero learns these probability distributions and expected outcomes entirely from self-play. In each position, an MCTS search is executed, guided by the neural network. Each search consists of a series of simulated games of self-play that traverse a tree from root state until a leaf state is reached. It returns a probability distribution over moves. The triples of state, move distribution and winner are then sampled in order to train the neural network [19].

Most recently, Goldwaser and Thielscher have applied deep reinforcement learning to GGP by extending the AlphaZero algorithm [11]. To this end, they provided methods to deal with the limiting assumptions that AlphaZero makes about games that it plays [11].

1. To address the non zero-sum games, the expected outcome output of the neural network, i.e., 1 for a win, 0 for a draw, -1 for a loss, is replaced with an expected reward that is between 0 and 1. With this, each agent will simply try to maximize their reward with no need to consider the others. In this way, cooperative policies can be learned.
2. To deal with the asymmetric games, it uses a separate neural network for each player which is trained separately, and combines the early layers of all the players' neural networks since very similar features from the game state are extracted in these neural networks.
3. To cope with simultaneous games with multi-player, the transition from one state to another is no longer through a certain action of a certain player, but through the sequence of actions of all players in that turn. In order to generalize the turn-based games to the simultaneous, an action without any effect called "noop" is added. Thus, a player can only do "noop" when it is not her turn.
4. To remove the reliance on a board and a handcrafted neural network for each game, it uses the propositional network [5,10] as the input to the neural network. To this end, they take all nodes from the propositional network and convert them to vectors before feeding them into the network. The architecture of the neural network in [11] is shown in Fig. 2. That input is then passed through some automatically generated fully connected layers as follows: first all inputs pass through to a series of fully connected hidden layers, with each one half the size of the previous, finishing with a layer of size 50. Then each player's head comes off that layer as a common start, doubling the size of the layer with more fully connected layers until it reaches the number of legal actions. At this point, there is also a fully connected layer which goes to a single output for expected return prediction. All hidden layers use

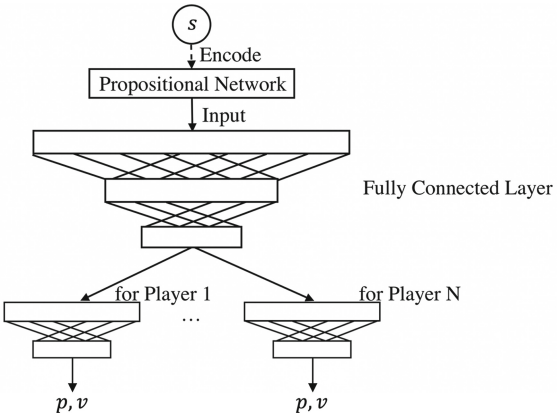

Fig. 2. The architecture of the neural network

ReLU activation, the expected reward uses sigmoid and policy output uses softmax. The output includes the probability distribution of this player p and the expectation of the final score of this player v in the state s.

To test the performance of their built learning agent, Goldwaser and Thielscher have conducted the comparison experiments with UCT as the benchmark due to its variants being state-of-the-art for many years. All hyperparameters were tuned on Connect-4 with a 6×7 board, and then evaluated on the typical games from past GGP competitions [8], namely Connect4, Babel, Breakthrough, Pacman3P. The experimental results show that with deep reinforcement learning the built agent performs better than the UCT benchmark agent in these games except the cooperative game Babel, as compared with their learning agent, the setup of the UCT agent tended to perform correlated actions which worked well as a joint strategy in Babel [11].

3 Method

In this section, we build a general game player by combining M-MCTS and deep reinforcement learning. First, we extend M-MCTS to suit the domain of GGP. Then we build a general game player by integrating the extension with deep reinforcement learning.

3.1 M-MCTS for GGP

As we mentioned before, M-MCTS was limited to game Go since a handcrafted neural network architecture for this game was used [24]. For board games such as Go, it is natural to extract features from the boards, while for GGP, the types of games are various and thus the method used in [24] is not applicable. In addition, it does not involve multi-player and simultaneous games. To remove

these limitations, we extend M-MCTS in the following aspects so as to suit the domain of GGP.

1. To deal with the deep convolutional neural network specializing in Go,
   (a) we redefined the feature function based on the propositional network, since games in GGP are all defined by Game Description Language (GDL), and propositional networks serve as a natural graph representation of GDL [5,10]. A Propositional Network is a directed bipartite graph consisting of nodes representing propositions connected to either boolean gates, or transitions [5]. Based on this, similar to [11], we specify the feature of the state as the vector of the node corresponding to that state in the propositional network. In this way, each state can be specified by a different and uniquely corresponding vector.
   (b) we further modified the distance function over game states based on their vectors. As the dimension of the vector reflects the information about each state, the closer the bitwise comparison of the two vectors is, the more similar the information of their corresponding states should be. As shown in the **Algorithm**, given two vectors we did the bitwise comparison, and the proportion of vector differences is used as their similarity value after mapping to $[-1, 1]$ by a linear transformation.
2. To incorporate the multi-player and simultaneous games, we used the sum of the estimated score of action sequences including the other players' actions as the estimated score of the player's own action. Similarly, the "noop" action is also added to treat turn-based games as a special case of concurrent games.

Algorithm. Distance Function $d(\vec{s}, \vec{x})$

Input: Vectors $\vec{s}$,$\vec{x}$
Output: the similarity of $\vec{s}, \vec{x}$

```
n ← the dimensions of s⃗
t ← 0
for i = 1 → n do
  if s_i == x_i then
    t ← t + 1
  end if
end for
return 2 * (−t/n) + 1
```

3.2 Combining M-MCTS with Deep Reinforcement Learning

Let us now combine the extended M-MCTS with deep reinforcement learning for building a general game player.

To this end, we keep the memory structure and the update strategy of M-MCTS, and replace the random simulations with the output of the neural network. The architecture is shown in Fig. 3. Although the architecture of the proposed player is similar to that of [11], there are following main differences

between them: we used the extended M-MCTS algorithm instead of MCTS to generate the training set through self-play, since the former is proved to be better than the latter and thus the quality of the training set is supposed to be improved. As we will show in the experiments, this consequently speeds up the training efficiency. Moreover, to integrate with deep reinforcement learning, we also did the following modifications:

1. Besides the memory-based value approximation, the effects of the neural network were also considered in the selection function. Formally, the selection formula is adapted as follows:

$$\mathbf{argmax}_{a \in A(s)} \left(\lambda_s * \hat{V}_{\mathcal{M}}(\delta(s,a)) + (1-\lambda_s) * \hat{V}(\delta(s,a)) + C * v * \frac{\sqrt{N(s)}}{N(\delta(s,a))+1} \right)$$

   where v is referring to the output of the policy network, which means the expectation of the final score in state s.
2. To make the algorithm more robust and also encourage exploration [20], the Dirichlet noise was added to the root of the searching tree by replacing v with $((1-w_{noise}) * v + w_{noise} * d)$, where d is a random variable from Dirichlet distribution which is the same as [11] and w_{noise} is the weight of the noise.
3. The estimation value of a state generated by random simulations is replaced by the output of the neural network, which is more accurate and more efficient to be calculated.

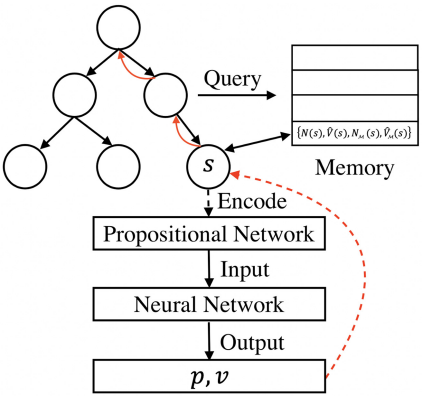

Fig. 3. The architecture of combining M-MCTS with deep reinforcement learning

4 Experimental Evaluation

In this section, we implement and evaluate the proposed GGP player with the benchmark UCT player which is commonly used in [6,8,11] in terms of a variety of games. We first introduce the evaluation methodology and the basic setting, and then provide the experimental results.

4.1 Evaluation Methodology

Similar to [11,12], we select the following four games in terms of complexity levels and game types to evaluate our built player extensively.

- Connect4 (6×7 board): two-player, zero-sum, player-symmetric, turn-based
- Breakthrough (6×6 board): two-player, zero-sum, player-symmetric, turn-based
- Babel: three-player, cooperative, player-symmetric, simultaneous
- Pacman 3P (8×8 board): three-player, cooperative/zero-sum, player-asymmetric, mixed turn-based/simultaneous

As we mentioned before, we also used the UCT player as the benchmark. Here we mainly focused on two aspects: the number of games trained on and the number of simulations per move during training. Both players had a limit of 1000 simulations per move during playing.

All hyperparameters were tuned on Connect4, and then evaluated on the other games. Here are the main parameters set in our experiments:

- $\mathcal{M}$, the size of memory structure, is set to 20.
- M, the top M similar states during the query, is set to 5.
- λ_s, the usage of memory, is set to 0.5.
- C, the parameter in UCB, is set to 2.
- w_{noise}, the weight of Dirichlet noise, is set to 0.25.

All the experiments were conducted on a server with Intel Xeon Gold 5218 running at 2.30 GHz.

4.2 Results and Analysis

Connect4. Our player scores 1 point for a win, 0.5 points for a draw, and no point for failure. As shown in Fig. 4, the performance of our player on Connect4 is better than that of UCT when selecting appropriate parameters and carrying out a certain amount of training. The winning rate can reach nearly 60% when the number of simulations per move during training is set to 100 and the number of games trained on is up to 4000. Given the same number of games trained on, the performance is much better when the number of simulations per move is 100 than 300.

Breakthrough. In this game, our player scores 1 point for a win and no point for failure. The results are shown in Fig. 5. The performance of our player on Breakthrough far exceeds that of the UCT. Especially, when the number of simulations per move during training is 300 and the number of games train on is up to 400, the winning rate is up to 95%.

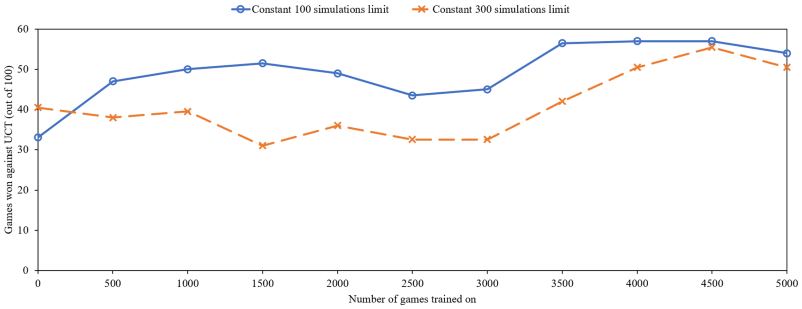

Fig. 4. Our player VS UCT in Connect4

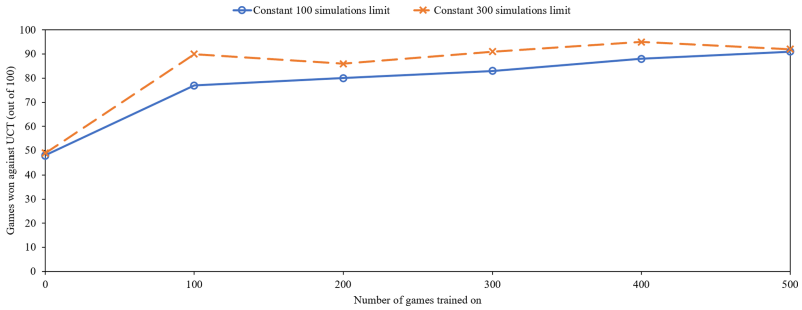

Fig. 5. Our player VS UCT in Breakthrough

Babel. There are three players in this game working together to build a tower. In a round, each builder had the same score, and we recorded the total score of each builder over 50 rounds. Each builder was given a score based on how well they reached the building level in one round. Each run consists of either three our players or three UCT players with no mixed teams. The results are shown in Fig. 6. Similar to the experimental results in [11], UCT players have performed much better on Babel. One of the most possible reasons is that the consistency policy which UCT players make has a great advantage on Babel. On the other hand, since we train the network individually for each role, it might not be easy to form a consistent policy for them. In addition, it can be found that our player performs better when the number of simulations per move during training is set to 300 rather than 100. It is likely that with some extra complexity in the neural network architecture and more training time it would be able to achieve a better performance.

Pacman 3P. In this game, one player controls Pacman to collect pellets and the other two players, each controlling a Ghost, work together to catch Pacman. In one round, for Ghosts, catching Pacman scores 1 point; for Pacman, there are 35 pellets, catching one pellet scores 1/35 points. For the evaluation, it was played both with Pacman being a UCT agent and both ghosts being our players

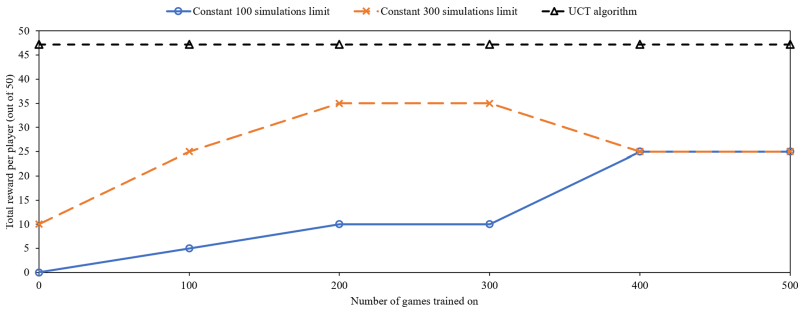

Fig. 6. Our player VS UCT in Babel

and the reverse. The results in terms of the number of simulations 100, 300 per move during training are shown in Fig. 7 and Fig. 8, respectively. As we can see, our player playing as Pacman and Ghosts can perform better than UCT when the number of simulations per move during training is set to 100. Yet when the number of simulations per move during training is increased from 100 to 300, the performance of our player shows a tendency of decline.

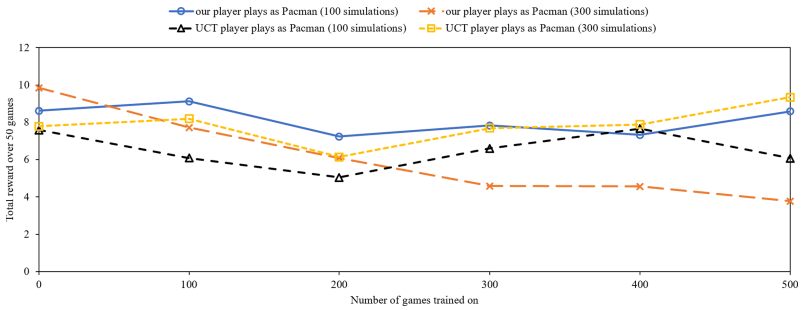

Fig. 7. Our player and UCT player play as Pacman in Pacman3P.

To conclude this section, on one hand, the experimental results show that our player with appropriate parameters and sufficient training can significantly outperform the benchmark UCT player on a number of games in both efficiency and effectiveness except the cooperative game Babel, which is similar to [11,12]. On the other hand, we also find that regarding the two parameters we concern it is not the case that the larger they are, the better our player performs. For instance, the performance of our player in game Pacman 3P shows a tendency of decline as the number of simulations per move during training increases. One possible reason is that at the very beginning the neural network could not make the right decision over actions due to the inaccurate probability distribution over actions and insufficient training data, which in turn might guide M-MCTS to

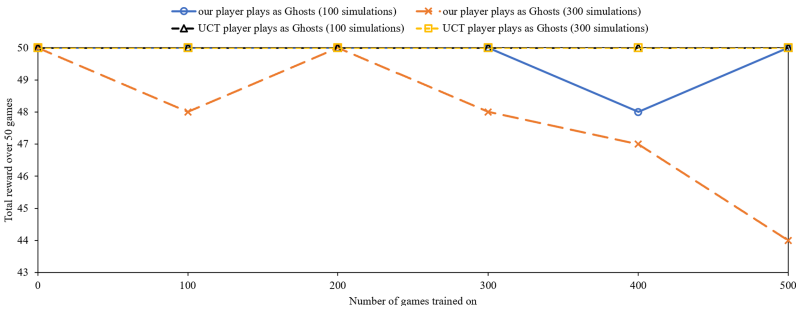

Fig. 8. Our player and UCT player play as Ghosts in Pacman3P.

generate the defective training set. This could make the neural network less effective than what we expect. Finally, it should be noted that without the details of their parameters, in this paper we did not conduct a direct comparison with Goldwaser and Thielscher's player. Yet as we can see from the above experimental results, to some extent, our algorithm has better results and fewer training rounds compared with theirs. A direct and detailed comparison is left for future work.

5 Conclusion

In this paper, we have extended Memory-Augmented Monte Carlo Tree Search algorithm (M-MCTS) to the domain of general game playing and integrate this extension with deep reinforcement learning for building a general game player. The experimental results have showed that it outperforms the benchmark UCT player in a variety of games except Babel. This work has confirmed the feasibility of applying M-MCTS and deep reinforcement learning for general game playing.

Directions of future research are manifold. This paper mainly explored the impacts of the number of games trained on and the number of simulations per move during training on the performance of our player. Besides the parameters we tuned on Connect4, we believe that the parameters related to M-MCTS algorithm we set will also affect the performance of our player and thus worth further investigating.

Regarding the feature function for M-MCTS, in this paper we coded a game state as a unique vector based on the propositional network. This method is natural and works well to extend M-MCTS from single game Go to various types of games. Yet it would be interesting to investigate other methods to improve the feature function so as to make the similarity of the states more accurate.

Last but not least, it is worth noting that our player is better than the UCT player in the simulation efficiency, especially at the beginning of a game situation. To further improve the performance of the player, we believe that larger and deeper networks for games are a potential approach, since it has been shown in [12] that deeper networks with more training tended to have better performance than shallower networks.

Acknowledgments. We are grateful to the reviewers of this paper for their constructive and insightful comments. The research reported in this paper was partially supported by the National Natural Science Foundation of China (No. 61806102), the Major Program of the National Social Science Foundation of China (No. 17ZDA026), and the National Key Project of Social Science of China (No. 21AZX013).

References

1. Auer, P.: Using confidence bounds for exploitation-exploration trade-offs. J. Mach. Learn. Res. **3**, 397–422 (2002)
2. Ayuso, J.L.B.: Integration of general game playing with RL-glue (2012). http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.224.7707&rep=rep1&type=pdf
3. Brown, N., Sandholm, T.: Libratus: the superhuman AI for no-limit poker. In: Proceedings of the 26th International Joint Conference on Artificial Intelligence, pp. 5226–5228 (2017)
4. Clune, J.: Heuristic evaluation functions for general game playing. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence, pp. 1134–1139 (2007)
5. Cox, E., Schkufza, E., Madsen, R., Genesereth, M.: Factoring general games using propositional automata. In: Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents, pp. 13–20 (2009)
6. Finnsson, H., Björnsson, Y.: Simulation-based approach to general game playing. In: Proceedings of the 23rd AAAI Conference on Artificial Intelligence, pp. 259–264 (2008)
7. Finnsson, H., Björnsson, Y.: Learning simulation control in general game-playing agents. In: Proceedings of the 24th AAAI Conference on Artificial Intelligence (2010)
8. Genesereth, M., Björnsson, Y.: The international general game playing competition. AI Mag. **34**(2), 107–107 (2013)
9. Genesereth, M., Love, N., Pell, B.: General game playing: overview of the AAAI competition. AI Mag. **26**(2), 62–72 (2005)
10. Genesereth, M., Thielscher, M.: General game playing. Synthesis Lectures on Artificial Intelligence and Machine Learning **8**(2), 1–229 (2014). https://doi.org/10.1007/978-981-4560-52-8_34-1
11. Goldwaser, A., Thielscher, M.: Deep reinforcement learning for general game playing. In: Proceedings of the 34th AAAI Conference on Artificial Intelligence, pp. 1701–1708 (2020)
12. Gunawan, A., Ruan, J., Thielscher, M., Narayanan, A.: Exploring a learning architecture for general game playing. In: Gallagher, M., Moustafa, N., Lakshika, E. (eds.) AI 2020. LNCS (LNAI), vol. 12576, pp. 294–306. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-64984-5_23
13. Hsu, F.H.: Behind Deep Blue: Building the Computer That Defeated the World Chess Champion. Princeton University Press, Princeton (2002)
14. Koriche, F., Lagrue, S., Piette, É., Tabary, S.: General game playing with stochastic CSP. Constraints **21**(1), 95–114 (2015). https://doi.org/10.1007/s10601-015-9199-5
15. Méhat, J., Cazenave, T.: A parallel general game player. KI-künstliche Intelligenz **25**(1), 43–47 (2011)
16. Moravčík, M., et al.: Deepstack: expert-level artificial intelligence in heads-up no-limit poker. Science **356**(6337), 508–513 (2017)

17. Schiffel, S., Thielscher, M.: Fluxplayer: a successful general game player. In: Proceedings of the 22nd AAAI Conference on Artificial Intelligence, pp. 1191–1196 (2007)
18. Silver, D., et al.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
19. Silver, D., et al.: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science **362**(6419), 1140–1144 (2018)
20. Silver, D. et al.: Mastering the game of Go without human knowledge. Nature **550**(7676), 354–359 (2017)
21. Świechowski, M., Park, H., Mańdziuk, J., Kim, K.J.: Recent advances in general game playing. Sci. World J. **2015** (2015)
22. Thielscher, M.: General game playing in AI research and education. In: Bach, J., Edelkamp, S. (eds.) KI 2011. LNCS (LNAI), vol. 7006, pp. 26–37. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24455-1_3
23. Wang, H., Emmerich, M., Plaat, A.: Monte carlo Q-learning for general game playing. arXiv preprint arXiv:1802.05944 (2018)
24. Xiao, C., Mei, J., Müller, M.: Memory-augmented Monte Carlo tree search. In: Proceedings of the 32nd AAAI Conference on Artificial Intelligence (2018)