Evolving Policy Sets for Multi-Policy Decision Making

Maximilian Krogius[1] Edwin Olson[1,2]

***Abstract*— Multi-Policy Decision Making (MPDM) is a planning framework in which an agent dynamically switches between a set of policies by predicting the performance of those policies using forward simulation. But in virtually all MPDM approaches, the set of policies are created by domain experts.**

In this paper, we learn these policy sets off-line. We use an evolutionary algorithm approach, which allows us to directly optimize the performance of the policy set, rather than some proxy objective.

We also propose the use of Terminal, an online strategy game, as an evaluation domain for planning algorithms. Like many real-world robotics problems, Terminal requires multi-agent planning, coping with uncertainty, and practical limits on computational complexity. We describe how we used our approach to generate an agent which is ranked in the top 10 in a global online competition.

I. Introduction

A fundamental problem in robotics is that of deciding how to move or act in the world. A common approach from controls or reinforcement learning is to optimize a policy function. This policy function maps from states to actions, determining the robot's action in all possible situations.

However, it is not clear that this sort of monolithic policy is the right approach for a more complex environment. Instead it may be better to split up the robot's behavior into simpler policies and switch between these policies when needed. For example, for a robot navigation task, the robot could follow the right-hand wall to travel along a corridor, and then switch to a slow but safe navigation policy to pass through a narrow door. Having multiple simpler policies allows generalization of the robot's behavior because it can mix and match its existing set of policies to fit new situations.

Unlike learning a single policy, learning a set of policies poses a unique challenge. Ideally each policy in the set would encode a different behavior, with the overall set of policies being mutually complementary. This can be difficult to achieve if each policy is trained individually with the same objective of increasing performance because there is no signal to a policy that it should learn a policy that is different from previously-learned policies.

Inspired by recent work on policy evolution, we propose to evolve our multi-policy system. This removes any need to specify objectives for the individual policies. Instead, we can evaluate the fitness of each multi-policy agent as a whole, correctly rewarding agents which have a set of mutually complementary policies.

[1]Robotics Institute, University of Michigan
[2]Computer Science and Engineering Department, University of Michigan
{mkrogius,ebolson}@umich.edu

Distribution A. Approved for public release; distribution unlimited. (OPSEC 4658)

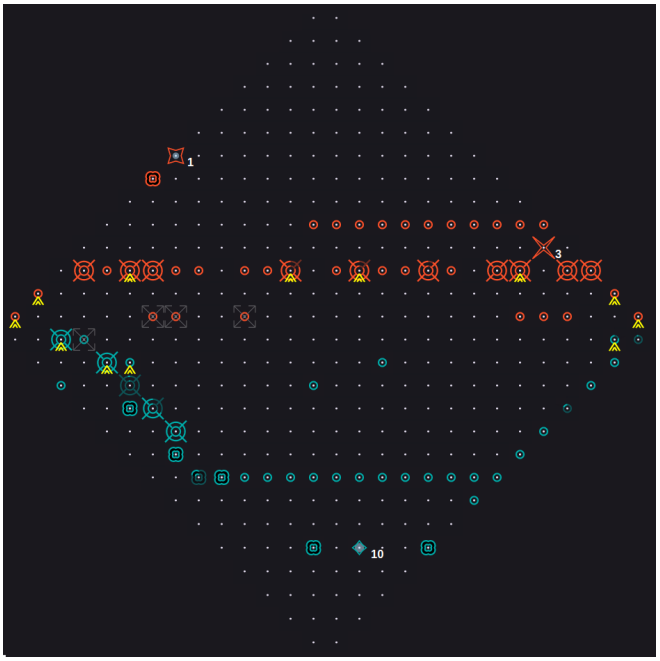

Fig. 1. A game of Terminal between our agent (bottom/green) and an opponent, online. The green objects are our agent's units and the orange ones belong to our opponent. Each turn, both players make their moves simultaneously, playing multiple stationary and mobile units. The stationary units remain on the board between turns, building up a defense over time while the mobile units attempt to cross to the far side of the board to score on the opponent. This game requires planning with a massive action space while the simultaneous play adds the difficulties of planning under uncertainty and multi-agent planning. All of this takes place under the additional constraint of six cpu core-seconds per turn.

There are multiple different approaches to multi-policy systems but we wanted to choose one which works well in real robotics applications. For this paper, we will use Multi-Policy Decision Making [1] and evolve the policies used within it. MPDM works by, at every time step, running forward simulations for every simple policy for a fixed horizon. The simulations are then evaluated with a value function, and the policy with the best value is then executed for that timestep.

We evolve our MPDM agents using an efficient formulation of regularized evolution [2] and tournament selection. The relative fitness of two agents is evaluated through self-play, meaning that only one match needs to be played per new mutated agent.

While the ultimate goal is to show that this method will be effective for robotic planning, since this is the first step in developing this learning method, we have chosen a somewhat simpler task to begin with. The task we use is a two-player

game called Terminal. It has the complexity of involving interaction with another agent (since it is a two-player game). In addition, both players make their moves simultaneously, which distinguishes it from other games such as chess or go. Finally, it also has limits on allowable compute of just six seconds per turn on a single cpu core. It also has an active community of people who develop algorithms to play the game, allowing us to compare against a a wide selection of systems created by domain experts.

The contributions of this work include:

- An efficient method for evolving agents for two-player games through self-play.
- Initialization of the learning process with policy sets scraped from games played online by other players.
- The introduction of Terminal as a testbed for planning methods and the application of MPDM to this problem.
- A demonstration of the effectiveness of our approach where our evolved agent achieved rank 10 in a global online competition against other algorithms from competitors around the world.

II. Related Work

As our method combines policy learning, evolution, and planning we will mention all these topics in our related works.

A. Deep RL Policy Learning

Some of the most prominent works in policy learning from the past few years come from the field of deep reinforcement learning. The first example is AlphaGo [3] which was the first algorithm to achieve superhuman performance in the game of Go. Shortly thereafter, NFSP [4] showed that reinforcement learning could be applied to imperfect information games, achieving performance on par with other top computer programs at Limit Texas Hold'em poker. Finally, AlphaStar [5] achieved performance greater than 99.8% of human players at the game of Starcraft II. Based on the success of these methods we can assume that Terminal should be tractable with deep reinforcement learning. However, the game of Terminal has the additional restriction that moves must be calculated in less than six seconds on a single CPU core, which makes it unlikely that solutions based on large neural nets could be competitive.

B. Options

Options are an idea with a long history in reinforcement learning [6]. The idea is to learn multiple closed loop policies and to use these policies to enable temporally extended planning. This concept is similar to the idea of the simple policies in an MPDM system, where in both cases the policies are handcrafted. More recently, the option-critic architecture [7] was proposed, which uses policy-gradient-based optimization to learn a set of options from scratch. They showed that this multi-policy approach can outperform a single-policy approach using the DQN framework. Compared to our method, the use of gradient-based optimization allows for efficiently learning neural network based policies from scratch. However, our method does have the theoretical advantage that it is directly optimizing the performance of the system as a whole, instead of optimizing a proxy objective such as the performance of a single policy. In their following work on option-critic [8] they show that adding an additional term to the objective function representing a deliberation cost is necessary in order to learn options that are useful for a longer period of time.

C. Evolutionary Algorithms

The MAP-Elites [9] algorithm is an evolutionary algorithm that has been used in robotics to allow a robot's control policy to adapt to damage [10]. Offline evolution is used to create a set of policies and then an online trial and error algorithm learns which policy to use when the robot is damaged.

Evolutionary methods have been applied to learn a neural net based policy for the Atari Learning Environment [11], showing competitive performance with reinforcement learning methods. This work showed the authors that evolution could be a competitive learning method in complex environments. In this work we apply evolution to a game which has the additional complication of being a two-player instead of single-player game. Another evolutionary approach evolves the parameters of an agent while modelling the fitness landscape with an N-tuple system [12] for a single-player game. One approach similar to ours optimizes an agent for the game of Hearthstone [13]. Like our method, this tackles a two-player game using an evolutionary algorithm. Similar to them, we play head to head games between agents in the population in order to determine relative fitness, however we play only one game between agents, trading off noise of the fitness function for reduced compute cost per iteration.

D. Planning with Action Abstractions

A method called Puma [14] plans using macro-actions instead of simple policies. The macro-actions are not handcrafted, instead they are generated using sub-goals of attaining immediate reward and gaining information. Another method which uses online planning based on handcrafted policies is Portfolio Online Evolution in Starcraft [15]. This work uses an evolutionary algorithm in order to do online planning for starcraft.

Our planning method uses MPDM, which was originally introduced in the context of planning for self-driving vehicles [1]. In that work, MPDM was used to plan the actions of a car using a set of hand-crafted policies. Following works [16], [17] have also used handcrafted policies, although Mehta [18] has used online optimization of continuous policy parameters. Keesmat [19] is the first use of an evolutionary method alongside MPDM. They use a short evolutionary run to learn the parameters of a social force model used inside MPDM, as well as parameters relating to the MPDM system as a whole such as replanning interval and simulation horizon. Compared to this work we evolve policies with many more parameters, and run the evolution

for much longer because we are trying to tackle a complex two-player game.

III. Methods

A. Evolution

Our evolutionary method is based on regularized evolution [2]. Every iteration, we select two agents at random out of a population of at most 5000 agents. The relative fitness of these two randomly chosen agents is then determined by playing one game of Terminal between them. We mutate the winner of this game and add this copy to the population. We also save the winning algorithm to disk, for later evaluation. Lastly, if the population has reached maximum size, we remove the oldest agent in the population. This process is then repeated as many times as desired, up to 4 million iterations for the largest run. In practice, 96 iterations are run in parallel, so that we can use all cores of the server the code is running on.

Algorithm 1: Evolutionary Algorithm.

```
Given:
      n_iters // Maximum number of iterations
      max_size // Maximum population size
      scraped_policies // Initial policy sets
initialize population with scraped_policies
for n_iters do
    agent1 = random_choice(population)
    agent2 = random_choice(population)
    winner = play_match(agent1, agent2)
    save winner to disk
    new_agent = mutate(winner)
    population.push_back(new_agent)
    if size(population) > max_size then
        population.pop_front()
end
```

The most notable thing about this approach is that only one game is played per iteration. This allows each iteration to run quickly, at the cost of being a somewhat noisy evaluation of fitness. This noise of the evaluation comes from the fact that Terminal, being a simultaneous action game, has a rock paper scissors quality to it where the overall better agent may lose some matches where it is countered by the opponent's strategy.

This speedy evaluation allows our evolutionary runs to proceed quickly and efficiently. The cost of the run to produce our final agent was under $50 and took approximately 2 days of compute time (using a c5a.24xlarge spot instance on AWS).

B. Terminal

Terminal is a board game that is mostly played online by algorithms designed by domain experts. The rules are more complicated than chess or go so we will present a simplified version here (full version [20]). Terminal is a game where, each turn, both players make their moves simultaneously. A player's move consists of placing a number of stationary and mobile units on their side of the board and then submitting their turn. There are six different types of units that can be placed, three stationary unit types and three mobile unit types to be placed on a 28×28 board. Once both players submit their moves, the turn is then resolved through a number of complex rules governing the behaviour of these units. Mobile units will move across the board following rules for pathing, as well as rules governing how and when they will shoot at the opposing player's units. If the mobile units reach the opposite edge of the board then they will score. The stationary units are used to influence the pathing of the opposing player's units, shoot at the opposing player's units when within range, and to shield the player's mobile units.

Terminal has three complicating factors compared to games such as chess or go. First, on each turn of the game, both players make their moves simultaneously. Second, each move consists of placing multiple (typically $\sim$10) units of 6 different types on a board which is larger than a chess or go board. Third, calculating what happens after both players submit their moves is a more involved, iterative computation, with calculations required for each unit's pathing and targeting at each timestep of the turn.

All of these added complexities of Terminal make it less tractable to typical methods such as tree search. The larger board plus the multiple actions per turn mean that the action space is orders of magnitude larger than chess or go (an estimate of a typical move's action space is 3×10^{11} (see footnote[1]) whereas chess has an upper bound of valid moves of 4×10^3 (see footnote[2])). The simultaneous actions mean that the actions for both players must be considered simultaneously, meaning that an action space of size 9×10^{22} would need to be searched if one were to use typical tree search methods here. And the final difficulty for tree search methods is that the amount of computation per turn is limited to a maximum of six seconds on a single CPU core.

C. MPDM for Terminal

One must define a set of simple policies, an evaluation function, and a maximum simulation depth in order to use MPDM for any task. For this task we use a simulation depth of one, i.e. we simulate one turn and then evaluate the results. The evaluation function is a hand-designed nonlinear combination of each player's resources, and whether the player wins on this turn. Since our defensive structure is fixed, there are no terms in the evaluation function that are based on the defensive layout of either player.

The simple policies are designed to be highly parameterized so that their performance can be improved through evolution. To start, we factor each policy into a defence policy and an attack policy. A defense policy's parameters

[1]On a typical move the player can place 5 stationary pieces at at most $14 \times 28 \div 2$ locations. This gives an upper bound of $(14 \times 28 \div 2)^5 = 3 \times 10^{11}$.

[2]As a simple upper bound, a chess move consists of taking a piece from one position to another position on an 8×8 board. This gives an upper bound of $(8 \times 8)^2 = 4096$.

consist of a list of stationary units. These units will be built, in order, every turn, with as many units being built as the available cores will allow. Each set of policies that comprise an MPDM agent share a common defensive strategy but vary in terms of their attack. Note that different policy sets (within the evolutionary population) can have different defensive strategies. An attack policy's parameters consist of up to two locations to attack from, what types of units to attack with, and how much of the available bits should be allocated to each location. An attack policy will always attack using all available bits. In addition, there is also a default attack policy that does not attack at all.

MPDM chooses between these policies based on the result of simulations. Every turn, MPDM forward simulates the outcome for every simple policy and evaluates the outcome using the evaluation function. Then, the policy with the best evaluation is chosen and executed. During these simulations, a default policy is assumed for the opponent. This policy consists of the opponent building stationary units and scramblers that it has built on the previous turn or two.

Algorithm 3: MPDM for Terminal

```
Given:
    attack_policies
    defense_policy
    game // Represents the game state.

Function choosePolicy(game):
    for attack_policy ∈ attack_policies do
        sim = initialize_simulation(game)
        attack_move = evaluate_policy(attack_policy,
            sim)
        defense_move =
            evaluate_policy(defense_policy, sim)
        submit_turn(sim, move)
        simulate_turn(sim)
        value = calculate_endstate_value(sim)
        if value > best_value then
            best_value = value
            best_move = move
    end
    return best_move
end

while game is not over do
    best_move = choosePolicy(game)
    submit_turn(game, best_move)
end
```

D. Policy Scraping

We have used policy scraping to jump start the training and reduce the number of iterations required to produce a strong agent. We used policies scraped from games played between other agents in the online competition. The scraping algorithm works as follows. For a defense policy, we simply put every stationary unit built into a list. For an attack policy, every time the agent built mobile units we take the two locations at which the most mobile units were built and construct one of our attack policies with those parameters.

This form of policy scraping deliberately does not concern itself with producing strong policies. The goal is to produce policies which are close, in parameter space, to strong policies. The evolutionary algorithm will handle the problem of finding those nearby strong policies.

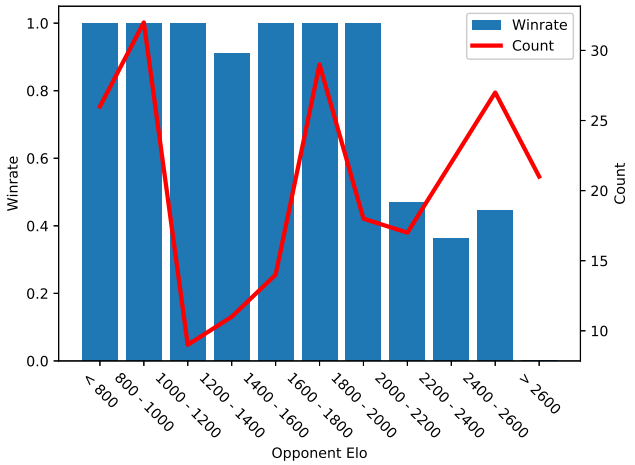

Fig. 2. Winrate of the final algorithm (100 policies), binned by opponent Elo. This shows all games played versus multiple versions of some opponents, not just the final end of season opponents. The agent performs quite well against lower or similarly rated opponents, but it cannot beat the opponents rated over 2600 Elo.

IV. Results

We ran our evolutionary method using an MPDM agent with 100 policies in order to produce our final agent. The starting agents were sampled from the scraped policies and evolved for 4 million iterations and took approximately 40 hours to complete on an AWS c5a.24xlarge spot instance. After completing the run, we ran a single elimination tournament between all agents that had managed to win at least one match during the evolutionary process. The winner of this competition was uploaded to the online competition.

Our agent came in at rank 10 out of 89 competitors for the final end of season ratings for season 6 of terminal. This competition includes a wide variety of agents designed by domain experts, so this result can be viewed as a comparison of our agent to a baseline of handcrafted agents.

The performance of our final agent vs different Elo levels is shown in Figure 2. This contains all matches played, not just the matches from the final end of season competition, broken out by the opponent's Elo rating. Our algorithm handily beats all opponents under 2200 elo, does slightly worse than 50% against agents between 2200 and 2600, and does not beat agents with rating over 2600.

We ran an ablation study in order to determine the effect of the number of policies as well as the effect of the use of scraped policies. The runs for this study were half that of the final run, i.e. 2 million iterations. In order to provide for

	1 Policy	5 Policies	15 Policies	50 Policies	50 Policies - No scraped policies
1 Policy	0.00	-0.50	-0.53	-0.56	0.72
5 Policies	0.50	0.00	-0.37	-0.25	0.86
15 Policies	0.53	0.37	0.00	0.05	0.60
50 Policies	0.56	0.25	-0.05	0.00	0.71
50 Policies - No scraped policies	-0.72	-0.86	-0.69	-0.71	0.00

TABLE I

The table shows the average score of each row's parameter settings vs the column's parameter setting. Score is defined as 1 for a win, 0 for a draw, -1 for a loss. The run without scraped policies is the weakest while the runs with 15 and 50 policies are the strongest.

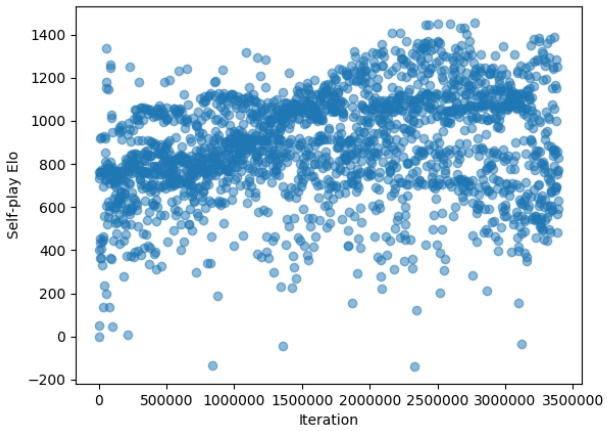

Fig. 3. The self-play elo of agents generated during the longest training run. The performance of the agents generated throughout the run is in general quite noisy, which is to be expected from an evolutionary algorithm. However, there is still gradual improvement being made as the best agents come from near the end of the training run. The self-play elo was determined by taking ∼2000 agents evenly sampled from throughout the training process, playing games between them, and using a bayesian Elo [21] approach to assign ratings.

a robust comparison between the different methods, the top 128 agents produced by each method were played against the top 128 agents produced by all other methods. These top 128 agent are selected by taking all the agents from the final 7 rounds of the single-elimination tournament.

In our ablation study, Table I, in almost every case the agent with more policies beats the agent with fewer policies, although the 50 policy and 15 policy agents seem to be of roughly equal strength. Additionally, the agents trained from scratch are far weaker than even the single policy agent trained from scraped policies. This shows that the use of many (at least 15) policies and the use of scraped policies for initialization are both important for the success of this method.

The training curve, as seen in Figure 3, shows how the performance of our agent grows over the course of the training run. While there are a few policies from early in the run that are high rated, the most highly rated policies only occur well into the run, at around 2.8 million iterations of training. While the training curve is noisy, it does show that the performance improves over time, eventually levelling off.

The self-play elo ratings for the training curve were generated first by sampling 1/1000th of the agents generated. Then the agents played matches against other agents of similar rating. Interleaved with this process, ratings were assigned to the agents using a minorization-maximization algorithm [21] in order to assign ratings in a bayesian fashion.

V. Conclusion

We have introduced Terminal as a testbed for robotic planning. This game has a massive action space and its simultaneous turn nature brings elements of multi-agent planning and planning under uncertainty. All of these challenges are relevant for robotic planning.

Our approach to Terminal uses an evolutionary algorithm to optimize our agent. We use a single self-play game to determine the fitness of agents during this process. This means that no external evaluation of the playing strength of an agent is required.

We can learn not just a policy, but a policy set since our learning method does not require differentiability. This is what allows us to learn the parameters of an MPDM system for Terminal and apply it in order to plan moves under strong time constraints.

Our system achieves rank 10 in a global online competition against algorithms written by domain experts around the world. This is a good result, but this also means that there are at least 9 agents that our agent cannot beat. We challenge the research community to attempt to produce an agent that can achieve first place in this competition.

References

[1] A. G. Cunningham, E. Galceran, R. M. Eustice, and E. Olson, "MPDM: Multipolicy Decision-Making in Dynamic, Uncertain Environments for Autonomous Driving," in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, June 2015.

[2] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized Evolution for Image Classifier Architecture Search," in *Proceedings of the aaai conference on artificial intelligence*, vol. 33, 2019, pp. 4780–4789.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the Game of go with Deep Neural Networks and Tree Search," *Nature*, vol. 529, pp. 484–489, 1 2016. [Online]. Available: https://www.nature.com/articles/nature16961

[4] J. Heinrich and D. Silver, "Deep Reinforcement Learning from Self-Play in Imperfect-Information Games," *CoRR*, vol. abs/1603.01121, 2016. [Online]. Available: http://arxiv.org/abs/1603.01121

[5] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster Level in StarCraft II using Multi-Agent Reinforcement Learning," *Nature*, vol. 575, pp. 350–354, 11 2019. [Online]. Available: https://www.nature.com/articles/s41586-019-1724-z

[6] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and Semi-MDPs: A Framework for Temporal Abstraction in Reinforcement Learning," *Artificial Intelligence*, vol. 112, pp. 181–211, 8 1999.

[7] P.-L. Bacon, J. Harb, and D. Precup, "The Option-Critic Architecture," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[8] J. Harb, P. Bacon, M. Klissarov, and D. Precup, "When Waiting is not an Option : Learning Options with a Deliberation Cost," *CoRR*, vol. abs/1709.04571, 2017. [Online]. Available: http://arxiv.org/abs/1709.04571

[9] J. Mouret and J. Clune, "Illuminating Search Spaces by Mapping Elites," *CoRR*, vol. abs/1504.04909, 2015. [Online]. Available: http://arxiv.org/abs/1504.04909

[10] A. Cully, J. Clune, D. Tarapore, and J.-B. Mouret, "Robots that Can Adapt Like Animals," *Nature*, vol. 521, no. 7553, pp. 503–507, 2015.

[11] F. P. Such, V. Madhavan, E. Conti, J. Lehman, K. O. Stanley, and J. Clune, "Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning," *CoRR*, vol. abs/1712.06567, 2017. [Online]. Available: http://arxiv.org/abs/1712.06567

[12] S. M. Lucas, J. Liu, and D. Perez-Liebana, "The N-Tuple Bandit Evolutionary Algorithm for Game Agent Optimisation," *2018 IEEE Congress on Evolutionary Computation, CEC 2018 - Proceedings*, 9 2018.

[13] P. García-Sánchez, A. Tonda, A. J. Fernández-Leiva, and C. Cotta, "Optimizing Hearthstone Agents Using an Evolutionary Algorithm," *Knowledge-Based Systems*, vol. 188, p. 105032, 1 2020.

[14] R. He, E. Brunskill, and N. Roy, "PUMA: Planning Under Uncertainty with Macro-Actions." in *AAAI*, 2010, p. 7.

[15] C. Wang, P. Chen, Y. Li, C. Holmgård, and J. Togelius, "Portfolio Online Evolution in StarCraft." in *AIIDE*, 2016, pp. 114–121.

[16] D. Mehta, G. Ferrer, and E. Olson, "Autonomous Navigation in Dynamic Social Environments Using Multi-Policy Decision Making," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 1190–1197.

[17] L. Zhang, W. Ding, J. Chen, and S. Shen, "Efficient Uncertainty-aware Decision-making for Automated Driving Using Guided Branching," *arXiv preprint arXiv:2003.02746*, 2020.

[18] D. Mehta, G. Ferrer, and E. Olson, "Backprop-MPDM: Faster Risk-Aware Policy Evaluation Through Efficient Gradient Optimization," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2018, pp. 1740–1746.

[19] P. Keesmaat, "Designing Socially Adaptive Behavior for Mobile Robots," Master's thesis, TU Delft, 2020.

[20] (2020) Terminal rules. [Online]. Available: https://terminal.c1games.com/rules

[21] D. R. Hunter *et al.*, "MM Algorithms for Generalized Bradley-Terry Models," *The annals of statistics*, vol. 32, no. 1, pp. 384–406, 2004.